Controlando versões com

Git e Github

ALEXANDRE AQUILES
RODRIGO FERREIRA

Sumário

1 Introdução **1**

1.1 Mantendo o histórico do código 1
1.2 Trabalhando em equipe 2
1.3 Sistemas de controle de versão 2
1.4 Controle de versão rápido e confiável com Git 3
1.5 Hospedando código no GitHub 3
1.6 O processo de escrita desse livro 4

2 Tour prático **5**

2.1 Instalando e configurando o Git 5
2.2 Criando um arquivo texto para versionarmos 7
2.3 Versionando seu código com Git 8
2.4 Compartilhando seu código através do GitHub 12

3 Trabalhando com repositório local **19**

3.1 Criando um repositório local 20
3.2 Rastreando arquivos 22
3.3 Gravando arquivos no repositório 29
3.4 Verificando o histórico do seu repositório 36
3.5 Verificando mudanças nos arquivos rastreados 39
3.6 Removendo arquivos do repositório 47
3.7 Renomeando e movendo arquivos 49
3.8 Desfazendo mudanças 52

4 **Trabalhando com repositório remoto** **61**
4.1 Repositório remoto 62
4.2 Adicionando o repositório remoto 63
4.3 Enviando commits para o repositório remoto 65
4.4 Clonando o repositório remoto 66
4.5 Sincronizando o repositório local 67
4.6 Protocolos suportados pelo Git 68

5 **Hospedando o repositório no GitHub** **71**
5.1 Serviços de hospedagem de projetos 72
5.2 GitHub: a rede social dos desenvolvedores 72
5.3 Encontrando projetos e visualizando o código-fonte 74
5.4 Criando um usuário no GitHub 80
5.5 Criando o repositório do projeto 83
5.6 Enviando os commits do projeto para o GitHub 85
5.7 Clonando o repositório hospedado no GitHub 86
5.8 Colaborando com projetos open source 87

6 **Organizando o trabalho com branches** **93**
6.1 A branch master 94
6.2 Criando uma branch 96
6.3 Trocando de branch 98
6.4 Deletando uma branch 100
6.5 Comitando código em uma nova branch 101
6.6 Voltando para o master e fazendo uma alteração 103
6.7 Mesclando alterações 105

7 **Trabalhando em equipe com branches remotas** **113**
7.1 Branches remotas 114
7.2 Compartilhando branches 116
7.3 Obtendo novas branches remotas em outros repositórios . . 118
7.4 Enviando commits para o repositório central 121
7.5 Obtendo commits de uma branch remota 123
7.6 Mesclando branches remotas e locais 127
7.7 Deletando branches remotas 133

8 Controlando versões do código com tags **137**
8.1 Criando, listando e deletando tags 138
8.2 Mais informações com tags anotadas 139
8.3 Compartilhando tags com a sua equipe 140

9 Lidando com conflitos **143**
9.1 Mesclando mudanças em um mesmo arquivo sem conflitos . 144
9.2 Conflitos após um merge com mudanças em um mesmo arquivo 146
9.3 Resolvendo conflitos após um rebase 150
9.4 Usando uma ferramenta para resolver conflitos 152

10 Maneiras de trabalhar com Git **157**
10.1 Utilizando só a branch master com um repositório central . 159
10.2 Utilizando branches por funcionalidade com um repositório central 162
10.3 Utilizando branches por etapa de desenvolvimento com um repositório central 167
10.4 Colaborando com projetos open source com Fork e Pull Request 173
10.5 Organizando projetos open source gigantescos com Ditador e Tenentes 175

11 Apêndice GitHub no Windows **179**
11.1 Instalando o GitHub for Windows 180
11.2 Criando um novo repositório 186
11.3 Efetuando commits no repositório 188
11.4 Detalhando os commits 190
11.5 Enviando o repositório para o GitHub 193
11.6 Trabalhando com branches 195

Versão: 17.5.7

Capítulo 1

Introdução

1.1 Mantendo o histórico do código

Vida de programador não é fácil. Há sempre uma pressão por entregas rápidas de novas funcionalidades. Mas, apesar da pressa, é necessário prestar atenção no que estamos fazendo, mesmo se a alteração for pequena. Ao mexermos em um código existente é importante tomarmos cuidado para não quebrar o que já funciona.

Por isso, queremos mexer o mínimo possível no código. Temos medo de remover código obsoleto, não utilizado ou até mesmo comentado, mesmo que mantê-lo já nem faça sentido. Não é incomum no mercado vermos código funcional acompanhado de centenas de linhas de código comentado.

Sem dúvida, é interessante manter o histórico do código dos projetos, para entendermos como chegamos até ali. Mas manter esse histórico junto ao código atual, com o decorrer do tempo, deixa nossos projetos confusos,

poluídos com trechos e comentários que poderiam ser excluídos sem afetar o funcionamento do sistema.

Seria bom se houvesse uma maneira de navegarmos pelo código do passado, como uma *máquina do tempo* para código...

1.2 Trabalhando em equipe

Mas, mesmo que tivéssemos essa máquina do tempo, temos outro problema: muito raramente trabalhamos sozinhos.

Construir um sistema em equipe é um grande desafio. Nosso código tem que se integrar de maneira transparente e sem emendas com o código de todos os outros membros da nossa equipe.

Como podemos detectar que estamos alterando o mesmo código que um colega? Como mesclar as alterações que fizemos com a demais alterações da equipe? E como identificar conflitos entre essas alterações? Fazer isso manualmente, com cadernetas ou planilhas e muita conversa, parece trabalhoso demais e bastante suscetível a erros e esquecimentos.

Seria bom que tivéssemos um *robô de integração* de código, que fizesse todo esse trabalho automaticamente...

1.3 Sistemas de controle de versão

Existem ferramentas que funcionam como *máquinas do tempo* e *robôs de integração* para o seu código. Elas nos permitem acompanhar as alterações desde as versões mais antigas. Também é possível detectar e mesclar alterações nos mesmos arquivos, além de identificar conflitos, tudo de maneira automática.

Essas ferramentas são chamadas de **sistemas de controle de versão**.

Nesse tipo de ferramenta, há um **repositório** que nos permite obter qualquer versão já existente do código. Sempre que quisermos controlar as versões de algum arquivo, temos que informar que queremos rastreá-lo no repositório. A cada mudança que desejamos efetivar, devemos armazenar as alterações nesse repositório.

Alterações nos mesmos arquivos são mescladas de maneira automática sempre que possível. Já possíveis conflitos são identificados a cada vez que obtemos as mudanças dos nossos colegas de time.

Desde a década de 1990, existe esse tipo de ferramenta. Alguns exemplos de sistemas de controle de versão mais antigos são CVS, ClearCase, SourceSafe e SVN (que ainda é bastante usado nas empresas).

Em meados da década de 2000, surgiram sistemas de controle de versão mais modernos, mais rápidos e confiáveis, como Mercurial, Bazaar e, é claro, Git.

1.4 Controle de versão rápido e confiável com Git

O **Git** é um sistema de controle de versão que, pela sua estrutura interna, é uma máquina do tempo extremamente rápida e é um robô de integração bem competente.

Foi criado em 2005 por Linus Torvalds, o mesmo criador do Linux, que estava descontente com o BitKeeper, o sistema de controle de versão utilizado no desenvolvimento do *kernel* do Linux.

Hoje em dia, além do *kernel* do Linux, a ferramenta é utilizada em diversos outros projetos de código aberto. O Git também é bastante utilizado em empresas em todo o mundo, inclusive no Brasil.

Atualmente, conhecer bem como utilizar o Git é uma habilidade importante para uma carreira bem-sucedida no desenvolvimento de software.

1.5 Hospedando código no GitHub

Em 2008, foi criado o GitHub, uma aplicação Web que possibilita a hospedagem de repositórios Git, além de servir como uma rede social para programadores.

Diversos projetos de código aberto importantes são hospedados no GitHub como jQuery, Node.js, Ruby On Rails, Jenkins, Spring, JUnit e muitos outros.

1.6 O processo de escrita desse livro

A utilização do Git não é restrita apenas ao desenvolvimento de software, muitos administradores de rede, por exemplo, utilizam o Git para manter o histórico de evolução de arquivos de configurações em servidores.

Acreditem, até mesmo a escrita desse livro foi feita utilizando o Git!

Não apenas esse, mas todos os livros da editora **Casa do Código** utilizam o Git como ferramenta de controle de versão, para manter o histórico de evolução dos capítulos. O GitHub também é utilizado para hospedagem dos repositórios dos livros.

Capítulo 2

Tour prático

Neste capítulo, faremos um *tour* bem prático sobre como usar o Git para versionar nossos projetos. Não se preocupe com o significado dos comandos. No decorrer do livro, todos os comandos usados aqui serão explicados com profundidade.

2.1 Instalando e configurando o Git

Antes de utilizarmos o Git, é fundamental instalá-lo. Escolha a seguir o Sistema Operacional apropriado e mãos à obra!

Instalando no Windows

Acesse a seguinte URL, faça o download e instale a última versão disponível: http://msysgit.github.io/

A instalação é bastante simples. Escolha as opções padrão.

Serão instalados alguns programas, sendo o mais importante o Git Bash, que permite que o Git seja executado pela linha de comando no Windows.

Dê duplo clique no ícone do Git Bash e será aberto um terminal, com o seguinte *prompt* na linha de comando:

```
$
```

Esse prompt será seu amigo a partir de agora. Não tenha medo! Sempre que falarmos de terminal, estaremos falando do Git Bash.

Instalando no Mac

Baixe a última versão do instalador gráfico do Git para Mac OS X a partir do link: https://code.google.com/p/git-osx-installer/downloads

Abra um terminal e prepare-se para utilizar o Git!

Instalando no Linux

Para instalar o Git no Ubuntu, ou em uma outra distribuição baseada em Debian, execute em um terminal:

```
$ sudo apt-get install git
```

No Fedora, utilize:

```
$ sudo yum install git
```

Para as demais distribuições do Linux, veja o comando em: http://git-scm.com/download/linux

Configurações básicas

É importante nos identificarmos para o Git, informando nosso nome e e-mail. Em um terminal, execute os comandos a seguir:

```
$ git config --global user.name "Fulano da Silva"
$ git config --global user.email fulanodasilva.git@gmail.com
```

Claro, utilize **seu nome e e-mail**!

A linha de comando

A maneira mais comum de usar Git é pela linha de comando, acessível através de um terminal. É o jeito que a maior parte dos bons profissionais do mercado utiliza o Git e será nossa escolha nesse livro.

GitHub for Windows

A maioria dos usuários do Windows não tem o hábito de utilizar o prompt de comandos, e perfere instalar alguma aplicação visual para trabalhar com o Git.

Uma destas aplicações é o **GitHub for Windows**, e mostraremos como utilizá-la no capítulo 11.

2.2 Criando um arquivo texto para versionarmos

Antes de utilizarmos o Git, vamos criar na sua **pasta pessoal**, um diretório chamado `citacoes` com um arquivo `filmes.txt`.

Dentro do arquivo `filmes.txt`, coloque o seguinte conteúdo:

```
"Não há certezas, apenas oportunidades." (V de Vingança)
"Diga 'olá' para meu pequeno amigo!" (Scarface)
```

Pasta pessoal

A pasta pessoal (*ou home directory*, em inglês) é o local dos arquivos de usuário como documentos, fotos, músicas etc.

Se você não souber onde é a pasta pessoal, digite o seguinte comando em um terminal:

```
$ echo ~
```

No Windows Vista, 7 ou 8, será algo como `C:\Users\Fulano\` ou, no Git Bash, `/c/Users/Fulano/`.

No Windows 2000, XP ou 2003, será algo como `C:\Documents and Settings\Fulano\` ou, no Git Bash, `/c/Documents and Settings/Fulano`.

No Linux, será `/home/fulano` e no Mac OS X `/Users/Fulano`.

2.3 Versionando seu código com Git

Criando um repositório

Abra um terminal e vá até o diretório `citacoes`.

```
$ cd ~/citacoes
```

Para transformar o diretório atual em um repositório do Git, basta executar o comando `git init`:

```
$ git init
```

Deverá aparecer uma mensagem semelhante à seguinte:

```
Initialized empty Git repository in /home/fulano/citacoes/.git/
```

Pronto, o projeto já é um repositório Git vazio.

Observe que foi criada uma pasta oculta com o nome `.git`.

Rastreando o arquivo

Mas e o arquivo `filmes.txt`? Será que já está versionado?

Podemos ver a situação dos arquivos no repositório Git com o comando:

```
git status
```

A saída deverá ser algo como:

```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#    filmes.txt
nothing added to commit but untracked files present (use
"git add" to track)
```

Observe a mensagem anterior: ela indica que o arquivo `filmes.txt` ainda não foi rastreado pelo Git.

Para que o arquivo seja rastreado, devemos executar o seguinte comando:

```
git add filmes.txt
```

Agora, se executarmos `git status` novamente, teremos a seguinte saída:

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file:   filmes.txt
#
```

Gravando o arquivo no repositório

O resultado anterior mostra que o conteúdo do arquivo `filmes.txt` já está sendo rastreado pelo Git, mas ainda não foi gravado (ou *comitado*, em uma linguagem mais técnica) no repositório.

Para gravarmos as mudanças no repositório, devemos executar o comando:

```
git commit -m "Arquivo inicial de citacoes"
```

Observe que foi invocado o comando `git commit`, com a opção `-m` para informar a mensagem do commit.

Deve ter aparecido algo como a seguinte mensagem:

```
[master (root-commit) 3bc8267] Arquivo inicial de citacoes
 1 file changed, 2 insertions(+)
 create mode 100644 filmes.txt
```

Se executarmos `git status` novamente, teremos:

```
# On branch master
nothing to commit, working directory clean
```

Alterando o arquivo

Insira mais uma linha no arquivo `filmes.txt`, com o conteúdo:

```
"Hasta la vista, baby." (Exterminador do Futuro 2)
```

Depois disso, se executarmos `git status` novamente, podemos observar que há uma nova mudança para ser rastreada:

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in
working directory)
#
#    modified:   filmes.txt
#
no changes added to commit (use "git add" and/or
"git commit -a")
```

Rastreando e gravando as alterações no repositório

Para rastrearmos a modificação, devemos executar o comando `git add` novamente:

```
git add filmes.txt
```

Com a modificação rastreada, podemos gravá-la no repositório, com o comando `git commit`:

```
git commit -m "Inserindo nova citacao"
```

Devemos ter uma resposta parecida com:

```
[master 67f8a28] Inserindo nova citacao
 1 file changed, 1 insertion(+)
```

Verificando alterações realizadas

Para verificar o histórico das alterações gravadas no repositório, podemos executar o comando `git log`:

```
$ git log
```

A saída será parecida com:

```
commit 67f8a287ddd13021e85355694f0c42f0b4f8f6ff
Author: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Fri Apr 11 21:23:34 2014 -0300

    Inserindo nova citacao

commit 3bc8267d18d209bb38781aba69c29000d1e831cb
Author: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Fri Apr 11 21:21:38 2014 -0300

    Arquivo inicial de citacoes
```

Pronto! Temos um repositório criado com as alterações no arquivo `filmes.txt` devidamente gravadas. Mas e agora?

2.4 Compartilhando seu código através do GitHub

Para que o mundo possa descobrir nosso incrível projeto, temos que compartilhá-lo na internet. Para isso, utilizaremos uma aplicação web chamada GitHub.

Criando uma conta no GitHub

O primeiro passo é criar uma conta no GitHub. Para projetos de código aberto, não há custo nenhum! Com um navegador, acesse: https://github.com/

Preencha seu nome, e-mail e escolha uma senha.

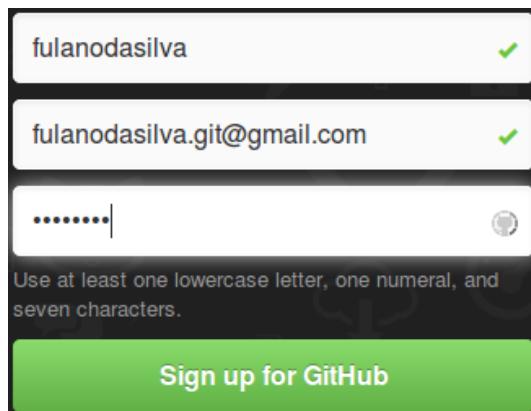



Figura 2.1: Criando conta no GitHub

Então, selecione o plano apropriado e finalize o cadastro, clicando em *Finish Signup*.

Figura 2.2: Selecionando plano no GitHub

Se necessário, verifique o e-mail.

Criando um repositório no GitHub

Agora podemos criar um repositório remoto, que ficará disponível para todos da internet. Para isso, clique no botão *New Repository* após acessar: https://github.com/

Figura 2.3: Novo repositório no GitHub

No *Repository name*, devemos preencher o nome do repositório remoto. No nosso caso, vamos preencher com "citacoes". Deixe o repositório como *Public*, para que qualquer pessoa consiga ver o seu código. As demais opções podem ficar com os valores padrão. Finalmente, devemos clicar em *Create repository*.

Figura 2.4: Criando repositório no GitHub

Pronto, já foi criado um repositório vazio lá no GitHub.

Apontando seu projeto para o GitHub

Devemos agora apontar o repositório da nossa máquina para o repositório do GitHub.

Em um terminal, certifique-se de estar no diretório `citacoes`, que tem o repositório local:

```
$ cd ~/citacoes
```

Então, execute o comando `git remote`, conforme o que segue:

```
$ git remote add origin https://github.com/fulanodasilva/
citacoes.git
```

Não deixe de **alterar** `fulanodasilva` para o **seu usuário do GitHub**. Não deve aparecer nenhuma mensagem.

Com o comando anterior, apontamos o nome `origin` para o repositório lá do GitHub.

Enviando as alterações para o GitHub

Com o repositório remoto configurado, podemos enviar nossas mudanças para o GitHub e, por consequência, para todo o mundo.

Para isso, basta executar o comando `git push`, da seguinte forma:

```
$ git push origin master
```

Com o comando anterior, enviamos as alterações para o repositório remoto configurado com o nome `origin`.

Forneça seu usuário e senha do GitHub quando solicitado. Deverá aparecer algo semelhante à seguinte saída:

```
Username for 'https://github.com': fulanodasilva
Password for 'https://fulanodasilva@github.com':
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 609 bytes | 0 bytes/s, done.
Total 6 (delta 1), reused 0 (delta 0)
To https://github.com/fulanodasilva/citacoes.git
 * [new branch]      master -> master
```

Vá até a página do seu projeto no GitHub: https://github.com/fulanodasilva/citacoes

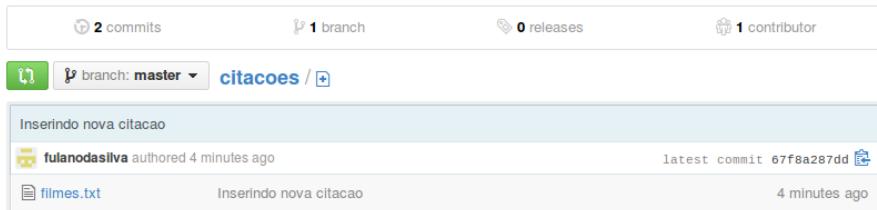



Figura 2.5: Página do projeto no GitHub

Observe que o arquivo que você enviou já está disponível para qualquer pessoa da internet. Avise seu primo, sua vizinha, todo mundo!

É possível ver todas as alterações no projeto até agora (no caso, foram duas), através do endereço: https://github.com/fulanodasilva/citacoes/commits/master

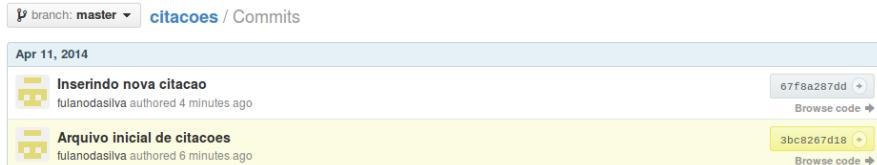



Figura 2.6: Listando alterações gravadas no GitHub

Se clicarmos na última alteração, por exemplo, é possível ver as mudanças que foram feitas. Fascinante, não?

Figura 2.7: Detalhando uma alteração gravada no GitHub

Obtendo projeto do GitHub

Com o projeto no GitHub, qualquer um pode acessar o código e ver o histórico, mesmo sem uma conta. Se a pessoa tiver cadastrada no GitHub, será possível baixar o código.

Vamos simular isso, baixando o código em outro diretório do seu computador.

Na sua pasta pessoal, crie um diretório chamado `projetos_git`. Então, o acesse pelo terminal:

```
$ cd ~/projetos_git
```

Para obter o código do projeto lá do GitHub, execute o comando `git clone` conforme o seguinte:

```
git clone https://github.com/fulanodasilva/citacoes.git
```

Não esqueça de **alterar** `fulanodasilva` para o **seu usuário do GitHub**.

Deverá aparecer algo como:

```
Cloning into 'citacoes'...
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 1), reused 6 (delta 1)
Unpacking objects: 100% (6/6), done.
Checking connectivity... done
```

Observe no diretório `projetos_git` que foi criado um subdiretório chamado `citacoes`. Dentro desse subdiretório há o arquivo `filmes.txt`, com exatamente o mesmo conteúdo lá do GitHub.

Há também um diretório oculto chamado `.git`, revelando que temos uma cópia do repositório original.

Vá até o subdiretório `citacoes`, executando o comando:

```
$ cd citacoes
```

Podemos executar o comando `git log` nesse novo repositório:

```
$ git log
```

Teremos a mesma saída de antes:

```
commit 67f8a287ddd13021e85355694f0c42f0b4f8f6ff
Author: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Fri Apr 11 21:23:34 2014 -0300

    Inserindo nova citacao

commit 3bc8267d18d209bb38781aba69c29000d1e831cb
Author: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Fri Apr 11 21:21:38 2014 -0300

    Arquivo inicial de citacoes
```

Se você tiver outro computador, faça o `clone` nele. Não se esqueça de instalar o Git, seguindo os passos do início deste capítulo.

Git e GitHub são a mesma coisa?

Não. Git é o sistema de controle de versões, com o qual interagimos na linha de comando. Já o GitHub é uma rede social para programadores que disponibiliza repositórios Git acessíveis remotamente. O GitHub é muito utilizado para projetos open source que possuem vários colaboradores do mundo todo. Mais adiante aprofundaremos em ambos.

Capítulo 3

Trabalhando com repositório local

A empresa Móveis Ecológicos nos contratou para fazer a página da empresa na web. Conversamos com o dono da empresa, discutimos sobre o conteúdo básico e criamos um HTML simples.

Apresentamos o resultado e nosso cliente pediu para colocarmos a página no ar. Logo o fizemos: diversão e lucro!

Porém, a diretora de relações públicas da empresa não gostou do texto. Pediu para alterarmos para um texto bem maior e com mais cara de *press release*. Fizemos as alterações e publicamos.

Depois de algum tempo, o dono da empresa notou o site modificado e nos contatou para voltarmos com a versão anterior. Acontece que esquecemos de fazer *backup* do arquivo. Até tentamos usar um recuperador de arquivos ou

lembrar o conteúdo de cabeça, mas não tivemos sucesso. E agora?

Com muita relutância, entramos em contato com o dono da empresa pedindo que o texto fosse dito novamente por telefone. Depois de alguns minutos de xingamentos e de quase perdermos o contrato, conseguimos o texto mais uma vez.

Poxa vida... Devíamos ter mantido o texto antigo comentado na página... Mas com tantas mudanças, em pouco tempo teríamos mais comentários do que código de fato.

Se tivéssemos utilizado um sistema de controle de versão como o Git, teríamos uma máquina do tempo para o nosso código. Aí, nossa vida seria mais fácil.

3.1 Criando um repositório local

Vamos dizer que, na nossa pasta pessoal, temos um diretório chamado `moveis`, com o arquivo `index.html` contendo o seguinte conteúdo:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <meta name="description" content="Móveis ecológicos">
        <meta name="keywords" content="moveis ecologicos">
    </head>
    <body>
        <h1>Móveis Ecológicos S. A.</h1>
        <ul>
            <li>Móveis de garrafas PET</li>
            <li>Móveis de latinhas de alumínio</li>
            <li>Móveis de papelão</li>
        </ul>
    </body>
</html>
```

Antes de tudo, vamos entrar no diretório `moveis`, com o comando `cd ~/moveis`.

Para criar um repositório Git nesse diretório, basta executarmos:

```
$ git init
```

Teremos a saída:

```
Initialized empty Git repository in /home/fulano/moveis/.git/
```

A mensagem anterior indica que transformamos o diretório `moveis` em um repositório Git.

Se listarmos os arquivos do diretório `moveis` com o comando `ls -lha`, teremos uma resposta semelhante ao que segue:

```
total 36K
drwxr-xr-x  3 fulano fulano 4,0K Abr 15 21:30 .
drwx------ 58 fulano fulano  16K Abr 15 21:28 ..
drwxr-xr-x  7 fulano fulano 4,0K Abr 15 21:30 .git
-rw-r--r--  1 fulano fulano  405 Abr 15 21:28 index.html
```

Observem que foi criado um subdiretório oculto chamado `.git` na pasta `moveis`. Esse subdiretório é um repositório do Git completo, que conterá todo o histórico de alterações dos arquivos, entre outras coisas.

> Podemos executar `git init moveis` se quisermos criar um diretório vazio que já é um repositório Git, ou seja, que já possui o `.git`.

No caso de termos outros subdiretórios como `js` para armazenar código Javascript e `css` para armazenar arquivos CSS, todas as informações desses arquivos serão armazenadas no mesmo `.git`.

Se por algum motivo quisermos parar de usar o Git, basta remover esse único diretório `.git`.

> Basta o comando `git init` para criar um repositório com Git, que já cria localmente um repositório completo. Isso é bem mais simples que diversos outros sistemas de controle de versão, como o SVN, que precisavam da configuração de um servidor.

3.2 Rastreando arquivos

Para verificarmos o estado atual do nosso repositório, devemos executar o comando:

```
$ git status
```

Deve aparecer uma resposta parecida com:

```
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#    index.html
nothing added to commit but untracked files present (use
"git add" to track)
```

Note que o arquivo `index.html` ainda não está sendo rastreado (está em *Untracked files*).

Para informar ao Git que o arquivo `index.html` deve ser rastreado, utilizamos o comando:

```
$ git add index.html
```

Pronto! Agora o Git sabe que esse arquivo é importante e as mudanças devem ser rastreadas.

Se executarmos novamente `git status`, teremos a seguinte saída:

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file:   index.html
#
```

Observe que o Git agora passou a rastrear o arquivo `index.html` e está pronto para gravar esse arquivo no repositório (*Changes to be committed*).

Rastreando vários arquivos

Vamos dizer que criamos um arquivo `estilos.css` no diretório `moveis`, com o seguinte conteúdo:

```
h1 {
  font-family: sans-serif;
}
li {
  font-family: monospace;
}
```

Também criamos um subdiretório `imagens` com um arquivo `logo.png`, com uma imagem semelhante a: http://goo.gl/6nganu

Se executarmos `git status`, teremos:

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file:   index.html
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#    estilos.css
#    imagens/
```

Observe que o arquivo `estilos.css` e o diretório `imagens` ainda não estão sendo rastreados (*Untracked files*).

Se quisermos rastrear todos esses arquivos, será que teremos que executar `git add` um por um?

Na verdade, não! Podemos rastrear todos esses arquivos de uma vez só com o comando:

```
$ git add .
```

O *ponto* do comando anterior representa todos os arquivos não rastreados do diretório atual e também de todos os seus subdiretórios.

Agora, ao executarmos `git status`, veremos que todos os arquivos estão sendo rastreados:

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file:   estilos.css
#    new file:   imagens/logo.png
#    new file:   index.html
#
```

Se quiséssemos ter rastreado apenas os arquivos do subdiretório `imagens`, por exemplo, poderíamos ter executado o comando `git add imagens`.

A área de stage

Quando informamos para o Git que queremos rastrear um arquivo, executando `git add` pela primeira vez, o Git coloca esse arquivo em uma área especial do repositório, chamada de *stage*.

Uma vez que um arquivo está na área de stage, todas as mudanças nesse arquivo passam a ser examinadas.

> O diretório que contém nossos arquivos é chamado de diretório de trabalho ou *working directory*, em inglês.

Vamos modificar o arquivo `index.html`, inserindo uma referência ao CSS criado anteriormente:

```
<!-- início do arquivo ... -->
    <head>
        <!-- tags meta ... -->
        <link rel="stylesheet" href="estilos.css">
    </head>
<!-- restante do arquivo ... -->
```

A saída do `git status` será:

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file:   estilos.css
#    new file:   imagens/logo.png
#    new file:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in
working directory)
#
#    modified:   index.html
#
```

Note que o `index.html` continua aparecendo como um novo arquivo rastreado (*new file* em *Changes to be committed*), indicando que ele está sendo rastreado e pronto para ser gravado.

Mas observe que o arquivo `index.html` aparece mais uma vez, sob *Changes not staged for commit*, como modificado (*modified*).

Isso acontece porque, uma vez que um arquivo passa a ser rastreado pelo Git, depois de colocá-lo na área de stage com o comando `git add`, cada

mudança nesse arquivo é rastreada e também deve ser colocada na área de stage.

Para colocarmos as últimas mudanças que fizemos no arquivo `index.html` na área de stage, devemos executar novamente:

```
$ git add index.html
```

Depois disso, ao executarmos `git status`, teremos:

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file:   estilos.css
#    new file:   imagens/logo.png
#    new file:   index.html
#
```

Agora, tanto o arquivo original como as últimas mudanças estão prontas para serem gravadas no repositório.

Uma representação gráfica das transições entre o diretório de trabalho e a área de stage seria:

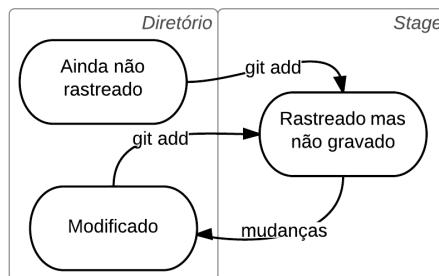



Figura 3.1: Rastreando arquivos e mudanças com Git

Ignorando arquivos

Suponha que temos um arquivo `todo.txt` que mantemos, durante o desenvolvimento, com a lista das tarefas do dia. Suponha também que temos um subdiretório chamado `tmp`, com alguns arquivos temporários gerados ao manipularmos imagens. Não faz sentido manter o histórico de ambos no Git.

Se executarmos `git status`, o arquivo `todo.txt` e o subdiretório `tmp` serão mostrados como arquivos ainda não rastreados:

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file:   estilos.css
#    new file:   imagens/logo.png
#    new file:   index.html
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#    tmp/
#    todo.txt
```

Será que teremos sempre que lembrar de evitar o `git add` desses arquivos? Se for assim, não poderemos mais utilizar `git add .`, já que esse comando rastreia todos os arquivos.

Para resolver isso, o Git tem um mecanismo que permite ignorarmos arquivos. Basta criarmos um arquivo chamado `.gitignore` no diretório principal do nosso projeto, com os nomes dos arquivos que queremos ignorar.

No nosso caso, devemos criar um arquivo chamado `.gitignore` no diretório `moveis`, com o seguinte conteúdo:

```
todo.txt
tmp/
```

Dessa maneira, o arquivo `todo.txt` e o subdiretório `tmp` não serão mostrados ao executarmos o comando `git status`:

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file:   estilos.css
#    new file:   imagens/logo.png
#    new file:   index.html
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#    .gitignore
```

Observe, porém, que o arquivo `.gitignore` apareceu como não rastreado. É importante que esse arquivo seja rastreado, porque evoluirá junto com o repositório. Por isso, ao criar o `.gitignore`, não esqueça de adicioná-lo à área de stage com o comando:

```
$ git add .gitignore
```

Se quiséssemos ignorar todos os arquivos com a extensão `.log`, por exemplo, colocaríamos `*.log` no `.gitignore`. Se quiséssemos ignorar todos os arquivos `.bmp` do subdiretório `imagens`, deveríamos inserir `imagens/*.bmp`.

Na prática, que tipo de arquivos são colocados no `.gitignore`? Depende da tecnologia utilizada no projeto.

Em projetos Java, arquivos `.class`, `.jar` e `.war` são exemplos de arquivos que devem ser ignorados. Para projetos Ruby, ignoraríamos arquivos `.gem` e o diretório `pkg`. Já para projetos Python, arquivos `.egg`.

Há um projeto no GitHub com exemplos de arquivos `.gitignore` para diversas linguagens de programação e tecnologias: https://github.com/github/gitignore

3.3 Gravando arquivos no repositório

Os arquivos e alterações que foram colocados na área de stage através do comando `git add` ainda não foram gravados no repositório. Ao invocarmos o `git status`, podemos notar que esses arquivos estão prontos para serem gravados (*Changes to be committed*):

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#    new file:   .gitignore
#    new file:   estilos.css
#    new file:   imagens/logo.png
#    new file:   index.html
#
```

Para gravar esses arquivos e alterações definitivamente no repositório, devemos utilizar o comando `git commit`:

```
$ git commit -m "Commit inicial"
```

Observe que passamos uma mensagem que descreve as alterações efetuadas, através da opção `-m`. Se não passássemos essa opção, seria aberto um

editor de texto para informarmos a mensagem.

No nosso caso, utilizamos a mensagem "Commit inicial". É importante que as mensagens descrevam de maneira sucinta as alterações que foram efetuadas.

Depois de executado esse comando, deverá aparecer algo como:

```
[master (root-commit) 4a4fdf7] Commit inicial
 4 files changed, 26 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 estilos.css
 create mode 100644 imagens/logo.png
 create mode 100644 index.html
```

Note que, na primeira linha da saída apresentada, é exibido um código logo antes da mensagem (`4a4fdf7`, no caso). Esse código serve como um identificador do commit. Na verdade, foram exibidos apenas os primeiros 7 dígitos desse código, que contém 40 caracteres ao todo.

> Mais especificamente, esse código é um número de verificação de integridade de dados (*checksum*) criado a partir do conteúdo do arquivo utilizando a função de *hash* SHA-1. Por utilizar SHA-1, é representado por um número hexadecimal de 40 dígitos.

Se verificarmos agora o estado do repositório com o comando `git status`, teremos:

```
# On branch master
nothing to commit, working directory clean
```

Observe que o não há nada mais a ser comitado.

> O termo *commit* é comum em qualquer sistema controlador de versão e significa gravar novos arquivos e alterações em arquivos existentes em um repositório. Em português, os termos *comitar* ou *comitado* são bem corriqueiros, apesar de não existirem nos dicionários.

Rastreando e comitando mudanças de uma só vez

Percebemos que havíamos esquecido de colocar o título na página `index.html`. Vamos alterar o arquivo:

```
<!-- início do arquivo ... -->
    <head>
        <!-- tags meta e link ... -->
        <title>Móveis Ecológicos</title>
    </head>
<!-- restante do arquivo ... -->
```

Além disso, vamos diminuir o tamanho da página para ficar melhor em dispositivos com telas pequenas. Para isso, alteraremos o arquivo `estilos.css`:

```
body {
    width: 50%;
    margin: auto;
}
/* resto do css */
```

Se executarmos `git status`, teremos:

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in
working directory)
#
#    modified:   estilos.css
#    modified:   index.html
#
no changes added to commit (use "git add" and/or
"git commit -a")
```

Note que os arquivos aparecem como modificados, mas a área de stage ainda não contém as mudanças efetuadas (*Changes not staged for commit*).

Poderíamos efetuar o `git add` dessas mudanças seguido do `git commit`. Porém, é possível rastrear as mudanças e comitá-las de uma vez com a opção `-a`:

```
$ git commit -a -m "Inserindo titulo e diminuindo tamanho da
pagina"
```

Teremos como resposta do comando anterior algo como:

```
[master ccff92c] Inserindo titulo e diminuindo tamanho da pagina
 2 files changed, 5 insertions(+)
```

A opção `-a` do comando `git commit` já efetua o rastreamento das mudanças, adicionando-as à área de stage. Poderíamos juntar as opções `-a` e `-m` utilizando `-am` da seguinte maneira: `git commit -am "Inserindo titulo..."`

Nesse ponto, a saída do comando `git status` indicará que não há mais nada a ser comitado:

```
# On branch master
nothing to commit, working directory clean
```

Novos arquivos precisam de git add

Nosso cliente pediu que colocássemos uma mensagem na página que é trocada automaticamente de tempos em tempos.

Para fazer isso, criamos um arquivo chamado `principal.js`, com o seguinte conteúdo:

```
var banners =
    ["Os melhores do Brasil!", "Qualidade e preço baixo!"];
var bannerAtual = 0;

function trocaBanner() {
  bannerAtual = (bannerAtual + 1) % 2;
  document.querySelector('h2#mensagem').textContent =
    banners[bannerAtual];
}

setInterval(trocaBanner, 2000);
```

Precisamos também adicionar o script anterior à nossa página `index.html`, além de uma tag `h2` que vai conter a mensagem:

```
<!-- início do arquivo ... -->
    <head>
        <!-- tags meta, link e title ... -->
        <script src="principal.js"></script>
    </head>
    <body>
        <!-- tag h1 ... -->
        <h2 id="mensagem"></h2>
<!-- restante do arquivo ... -->
```

Ao executarmos o comando `git status`, teremos a saída:

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working
directory)
#
#    modified:   index.html
#
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#    principal.js
no changes added to commit (use "git add" and/or
"git commit -a")
```

Observe que é mostrada a modificação no arquivo `index.html` e também o novo arquivo `principal.js`.

Vamos executar o comando para rastrear e comitar os arquivos de uma vez só:

```
git commit -am "Script de troca de banner"
```

Na saída, teremos:

```
[master bb229c6] Script de troca de banner
 1 file changed, 2 insertions(+)
```

Qual será a resposta do `git status`? Será:

```
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be
committed)
#
#    principal.js
nothing added to commit but untracked files present (use
"git add" to track)
```

Observe que o `git commit` com a opção `-a` adicionou no stage apenas as mudanças do arquivo que já estava sendo rastreado. O novo arquivo, `principal.js`, não passou a ser rastreado.

Sempre que tivermos um arquivo novo, temos que utilizar o comando `git add`, para que o Git passe a rastreá-lo. Só então poderemos comitá-lo.

Vamos lá:

```
$ git add principal.js
```

Após executarmos o `git status`, teremos:

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    new file:   principal.js
#
```

Agora podemos comitar o arquivo `principal.js`:

```
$ git commit -m "Inserindo arquivo principal.js"
```

Teremos uma saída parecida com:

```
[master 842d2cf] Inserindo arquivo principal.js
 1 file changed, 10 insertions(+)
 create mode 100644 principal.js
```

Ao executarmos o comando `git status`, teremos:

```
# On branch master
nothing to commit, working directory clean
```

Agora sim! Tanto a alteração em `index.html` como o novo arquivo `principal.js` estarão comitados no repositório.

Ao visualizarmos graficamente as transições entre o diretório de trabalho, a área de stage e o repositório propriamente dito, teríamos algo como:

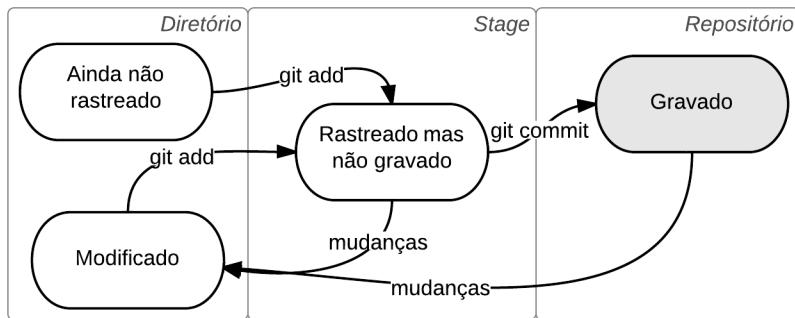



Figura 3.2: Gravando arquivos e mudanças com Git

Para saber mais: para que serve a área de stage?

O Git, ao contrário da maioria dos sistemas de controle de versão, possui uma separação entre rastrear as mudanças, adicionando-as na área de stage com o comando `git add`, e gravar as mudanças no repositório, com o comando `git commit`.

Mas será que essa separação é útil? É sim!

Vamos dizer que estamos desenvolvendo um sistema que tem um módulo de estoque e um de vendas. Ao desenvolver uma nova funcionalidade, modificamos quatro arquivos, sendo que dois são de estoque e dois são de vendas.

Poderíamos, primeiramente, gravar as alterações feitas nos dois arquivos de estoque, adicionando apenas esses dois à stage e comitando-os com uma mensagem bem descritiva. Depois, faríamos o `git add` e `git commit` dos arquivos de vendas, também com uma mensagem bem descritiva.

Ou seja, essa separação entre rastrear e gravar permite que as mudanças que fizemos no código sejam agrupadas de maneira lógica. Dessa forma, podemos montar commits menores, que farão mais sentido posterior ao revisarmos o histórico do projeto e ajudarão ao mesclarmos nossas mudanças com as dos outros membros do nosso time.

3.4 Verificando o histórico do seu repositório

Para verificar o histórico das mudanças gravadas no repositório, ou seja, os commits efetuados, devemos utilizar o comando:

```
$ git log
```

Após executar o comando anterior, teremos a seguinte saída:

```
commit 842d2cfc43418109e8f1bbd8ff8b91813f05373c
Author: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Wed Apr 30 14:32:38 2014 -0300

    Inserindo arquivo principal.js

commit bb229c6f5bf64f210fa5acb21e52286456e27ac8
Author: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Wed Apr 30 14:24:06 2014 -0300

    Script de troca de banner

commit ccff92c4036e50b573ad0a661a32416942da2c76
Author: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Wed Apr 30 14:02:26 2014 -0300

    Inserindo titulo e diminuindo tamanho da pagina

commit 4a4fdf77dc2e9e59c4b6a82523ab32dfc81859fc
Author: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Tue Apr 29 12:47:36 2014 -0300

    Commit inicial
```

Observe que foram listadas as mensagens de todos os commits, junto ao código correspondente, autor e data/hora em que foram efetuados.

Se quisermos mostrar apenas os dois últimos commits devemos utilizar a opção `-n`:

```
$ git log -n 2
```

Após executar o comando anterior, teremos a seguinte saída:

```
commit 842d2cfc43418109e8f1bbd8ff8b91813f05373c
Author: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Wed Apr 30 14:32:38 2014 -0300

    Inserindo arquivo principal.js

commit bb229c6f5bf64f210fa5acb21e52286456e27ac8
Author: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Wed Apr 30 14:24:06 2014 -0300

    Script de troca de banner
```

Se quisermos um resumo bem conciso dos commits do nosso projeto, podemos utilizar a opção `--oneline`:

```
$ git log --oneline
```

Será apresentado algo semelhante a:

```
842d2cf Inserindo arquivo principal.js
bb229c6 Script de troca de banner
ccff92c Inserindo titulo e diminuindo tamanho da pagina
4a4fdf7 Commit inicial
```

Podemos mostrar um resumo dos arquivos alterados, com o número de linhas adicionadas e removidas, através da opção `--stat`.

```
$ git log --stat
```

A saída será algo como:

```
commit 842d2cfc43418109e8f1bbd8ff8b91813f05373c
Author: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Wed Apr 30 14:32:38 2014 -0300

    Inserindo arquivo principal.js

 principal.js | 10 ++++++++++
 1 file changed, 10 insertions(+)

commit bb229c6f5bf64f210fa5acb21e52286456e27ac8
Author: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Wed Apr 30 14:24:06 2014 -0300

    Script de troca de banner

 index.html | 2 ++
 1 file changed, 2 insertions(+)

commit ccff92c4036e50b573ad0a661a32416942da2c76
Author: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Wed Apr 30 14:02:26 2014 -0300

    Inserindo titulo e diminuindo tamanho da pagina
:
```

Podemos observar que foram mostrados os arquivos modificados e o número de inserções.

Note também que o último caractere foi um dois pontos (`:`). Ao apertarmos a tecla `Enter`, serão mostrados os resultados restantes.

```
estilos.css | 4 ++++
 index.html  | 1 +
 2 files changed, 5 insertions(+)

commit 4a4fdf77dc2e9e59c4b6a82523ab32dfc81859fc
Author: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Tue Apr 29 12:47:36 2014 -0300

    Commit inicial
```

```
 .gitignore       |   3 +++
 estilos.css      |   6 ++++++
 imagens/logo.png | Bin 0 -> 15128 bytes
 index.html       |  20 ++++++++++++++++++++
 4 files changed, 29 insertions(+)
(END)
```

Para sairmos do resultados do `git log`, devemos apertar a tecla `q`.

Podemos também combinar as várias opções do comando `git log`. Por exemplo, para mostrar um resumo das alterações dos últimos dois commits:

```
$ git log -n 2 --oneline --stat
```

Teremos a seguinte saída:

```
842d2cf Inserindo arquivo principal.js
 principal.js | 10 ++++++++++
 1 file changed, 10 insertions(+)
bb229c6 Script de troca de banner
 index.html | 2 ++
 1 file changed, 2 insertions(+)
```

> Qual a diferença entre os comandos `git status` e o `git log`?
>
> O `git status` exibe arquivos que estão fora da área de stage, prontos para serem adicionados, e arquivos que estão dentro da área de stage, prontos para serem comitados.
>
> Já o `git log` exibe o histórico das mudanças efetivamente gravadas em um repositório. Ou seja, os commits efetuados.

3.5 Verificando mudanças nos arquivos rastreados

Percebemos um *bug* no nosso código: ao abrirmos a página, não é mostrado nenhum *banner*. Para corrigir, devemos invocar a função `trocaBanner` no evento `onload` da página, fazendo a seguinte mudança no arquivo `index.html`:

```
<!-- início do arquivo ... -->
    <body onload="trocaBanner();">
<!-- restante do arquivo ... -->
```

Depois de fazer essa alteração, ao invocarmos `git status`, teremos:

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working
directory)
#
#    modified:   index.html
#
no changes added to commit (use "git add" and/or
"git commit -a")
```

Tudo conforme o esperado. É informado que o arquivo `index.html` sofreu uma modificação.

Verificando mudanças ainda não rastreadas

Se quisermos revisar a modificação efetuada, verificando as diferenças entre o arquivo alterado e o que foi comitado anteriormente, podemos usar o comando:

```
$ git diff
```

Será mostrado algo como:

```
diff --git a/index.html b/index.html
index b1115c7..4786946 100644
--- a/index.html
+++ b/index.html
@@ -8,7 +8,7 @@
                <title>Móveis Ecológicos</title>
                <script src="principal.js"></script>
        </head>
-       <body>
+       <body onload="trocaBanner();">
```

```
            <h1>Móveis Ecológicos S. A.</h1>
            <h2 id="mensagem"></h2>
```

Observe que foi mostrada exatamente a mudança que fizemos anteriormente: a troca de `<body>` por `<body onload="trocaBanner();">`.

Quando tivermos alterações ainda não rastreadas em mais de um arquivo e quisermos verificar o que mudamos em um arquivo específico, basta passarmos o nome desse arquivo como parâmetro. Por exemplo, para verificarmos as mudanças apenas no arquivo `index.html`, faríamos: `git diff index.html`.

> O `git diff` não poderá ser utilizado para arquivos novos, que ainda não estão sendo rastreados pelo Git (ou seja, que ainda não tiveram o primeiro `git add` executado).

Verificando mudanças rastreadas

Vamos adicionar as mudanças que fizemos à área de stage com o comando `git add index.html`.

Se executarmos `git diff` novamente, não será mostrada nenhuma saída. O comando `git diff`, quando usado sem parâmetros, mostra a diferença entre os arquivos no diretório de trabalho e a área de stage. Portanto, serve apenas para exibir as mudanças ainda não rastreadas.

É possível mostrar as diferenças entre os arquivos na área de stage e a última versão que foi comitada utilizando a opção `--staged`:

```
$ git diff --staged
```

Será exibida uma saída exatamente igual à anterior, já que a alteração apenas passou para a área de stage mas continua a mesma:

```
diff --git a/index.html b/index.html
index b1115c7..4786946 100644
--- a/index.html
+++ b/index.html
@@ -8,7 +8,7 @@
```

```
                <title>Móveis Ecológicos</title>
                <script src="principal.js"></script>
        </head>
-       <body>
+       <body onload="trocaBanner();">
                <h1>Móveis Ecológicos S. A.</h1>
                <h2 id="mensagem"></h2>
```

> Antes da versão 1.6.1 do Git, só havia a opção `--cached`, que tem o mesmo efeito da opção `--staged`.

Verificando mudanças rastreadas e não rastreadas ao mesmo tempo

Vamos diminuir o tempo de troca do *banner* para um segundo alterando o arquivo `principal.js`:

```
//inicio do arquivo...
setInterval(trocaBanner, 1000);
```

Nesse ponto, temos duas alterações. Uma delas foi no arquivo `index.html` que já está na área de stage. Também temos uma alteração no arquivo `principal.js` ainda não rastreada. Isso pode ser verificado com o comando `git status`:

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    modified:   index.html
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working
directory)
#
#    modified:   principal.js
#
```

Se executarmos o comando `git diff`, veremos apenas a alteração no arquivo `principal.js`:

```
diff --git a/principal.js b/principal.js
index 5d114bd..696870a 100644
--- a/principal.js
+++ b/principal.js
@@ -6,5 +6,5 @@ function trocaBanner() {
   document.querySelector('h2#mensagem').textContent =
     banners[bannerAtual];
 }

-setInterval(trocaBanner, 2000);
+setInterval(trocaBanner, 1000);
```

Já ao executarmos o comando `git diff --staged`, veremos apenas a alteração no arquivo `index.html`, que já verificamos anteriormente.

Será que é possível exibir tanto as alterações fora da área de *stage* como as de dentro?

Sim! Para isso, precisamos descobrir o código do último commit. Podemos fazer isso com o comando `git log -n 1 --oneline`. Teremos a saída:

```
842d2cf Inserindo arquivo principal.js
```

Com o código do último commit em mãos, agora podemos mostrar as alterações dentro e fora da stage utilizando o comando:

```
$ git diff 842d2cf
```

Serão exibidas ambas as alterações:

```
diff --git a/index.html b/index.html
index b1115c7..4786946 100644
--- a/index.html
+++ b/index.html
@@ -8,7 +8,7 @@
                <title>Móveis Ecológicos</title>
                <script src="principal.js"></script>
```

```
        </head>
-       <body>
+       <body onload="trocaBanner();">
                <h1>Móveis Ecológicos S. A.</h1>
                <h2 id="mensagem"></h2>
diff --git a/principal.js b/principal.js
index 643ad98..696870a 100644
--- a/principal.js
+++ b/principal.js
@@ -6,5 +6,5 @@ function trocaBanner() {
   document.querySelector('h2#mensagem').textContent =
     banners[bannerAtual];
 }

-setInterval(trocaBanner, 2000);
+setInterval(trocaBanner, 1000);
:
```

Como o resultado é extenso, será mostrado o caractere dois pontos (`:`). Para irmos para os próximos resultados devemos pressionar a tecla `Enter`. Para sairmos, devemos apertar a tecla `q`.

> No nosso caso, teríamos a mesma saída anterior, que mostra as alterações dentro e fora do stage, utilizando o comando `git diff HEAD`.
>
> Isso acontece porque, no nosso caso, `HEAD` está apontado para o último commit efetuado.
>
> Porém, não é sempre assim, já que o `HEAD` pode apontar para commits anteriores.

Verificando mudanças já comitadas

Vamos comitar nossas modificações em `index.html` com o comando `git commit -m "Banner ao abrir pagina"`. Será exibido algo como:

```
[master 2d04ca8] Banner ao abrir a pagina
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Também precisamos comitar as alterações em `principal.js`, que ainda não está na área de stage. Para isso, utilizaremos a opção `-a`: `git commit -am "Diminuindo intervalo de troca de banner"`. Teremos na saída algo como:

```
[master 8aa07bd] Diminuindo intervalo de troca de banner
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Então, vamos mostrar os três últimos commits de maneira concisa através do comando `git log -n 3 --oneline`. A saída será semelhante a:

```
8aa07bd Diminuindo intervalo de troca de banner
2d04ca8 Banner ao abrir a pagina
842d2cf Inserindo arquivo principal.js
```

Agora não temos nenhuma alteração não comitada. Ufa!

Podemos usar o comando `git diff` para verificar as diferenças entre dois commits específicos. Para comparar o que foi alterado no nosso último commit em relação aos dois anteriores, devemos utilizar o `git diff` passando os códigos desses commits:

```
$ git diff 842d2cf..8aa07bd
```

Observe que passamos o código dos commits separados por ponto-ponto (`..`). Podemos ler o comado anterior como: "mostre as mudanças efetuadas a partir do commit 842d2cf até o commit 8aa07bd".

A resposta desse comando mostrará as alterações que acabamos de gravar no repositório:

```
diff --git a/index.html b/index.html
index b1115c7..4786946 100644
--- a/index.html
+++ b/index.html
@@ -8,7 +8,7 @@
                <title>Móveis Ecológicos</title>
                <script src="principal.js"></script>
        </head>
-       <body>
+       <body onload="trocaBanner();">
```

```
                <h1>Móveis Ecológicos S. A.</h1>
                <h2 id="mensagem"></h2>
diff --git a/principal.js b/principal.js
index 5d114bd..696870a 100644
--- a/principal.js
+++ b/principal.js
@@ -6,5 +6,5 @@ function trocaBanner() {
   document.querySelector('h2#mensagem').textContent =
     banners[bannerAtual];
 }

-setInterval(trocaBanner, 2000);
+setInterval(trocaBanner, 1000);
:
```

Poderíamos obter a mesma saída através do comando:

```
$ git diff 8aa07bd~2
```

O comando anterior exibe as mudanças nos arquivos do commit de código `8aa07bd` em relação aos dois commits feitos imediatamente antes. O número depois do `~` indica quantos commits anteriores devem ser considerados na comparação. No nosso caso, foram dois.

Apesar de apresentarem a mesma saída no nosso caso, os parâmetros `842d2cf..8aa07bd` e `8aa07bd~2` não são exatamente equivalentes.

Se houvesse alguma modificação ainda não comitada, fora ou dentro da área de stage, as linhas alteradas também seriam mostradas na resposta do comando `git diff 8aa07bd~2`.

Já o comando `git diff 842d2cf..8aa07bd` não exibe modificações ainda não comitadas, mas apenas as mudanças que aconteceram entre os dois commits especificados.

> O comando `git status` lista os arquivos modificados e o conteúdo da área de stage. Já o comando `git diff` mostra detalhadamente quais foram essas modificações, além de permitir verificar mudanças entre dois commits.

3.6 Removendo arquivos do repositório

Vamos dizer que criamos um arquivo `produtos.html` com o seguinte conteúdo:

```
<html>
    <body>
        <ul>
            <li>Móveis de garrafas PET</li>
            <li>Móveis de latinhas de alumínio</li>
            <li>Móveis de papelão</li>
        </ul>
    </body>
</html>
```

Então, informamos ao Git que queremos rastreá-lo, através do comando `git add produtos.html`. Depois, gravamos o conteúdo do arquivo no repositório com o comando `git commit -m "Página de produtos"`.

Mas acabamos percebendo que replicamos o conteúdo do arquivo `index.html` e, por isso, gostaríamos de remover o arquivo `produtos.html`.

Porém, não basta deletarmos o `produtos.html`. Precisamos deletá-lo e adicionar a deleção na stage, para só então efetuarmos um commit no repositório.

A remoção do arquivo e adição na stage podem ser realizadas de uma vez só através do comando:

```
$ git rm produtos.html
```

Depois do comando anterior, o arquivo `produtos.html` não existirá mais. Se executarmos `git status`, teremos:

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    deleted:    produtos.html
#
```

Observe que a deleção de `produtos.html` já está na área de stage, pronta para ser comitada com o comando `git commit -m "Removendo página de produtos"`.

Se já tivéssemos removido o arquivo `produtos.html`, ao executarmos `git rm produtos.html`, a deleção do arquivo seria colocada na área de stage.

Uma outra maneira de adicionar a deleção de um arquivo à área de stage seria executar o comando `git add`. Porém, esse comando não permite o uso em arquivos removidos, a não ser que seja utilizada a opção `--all`.

Adicionando a remoção de arquivos ao nosso gráfico, teríamos:

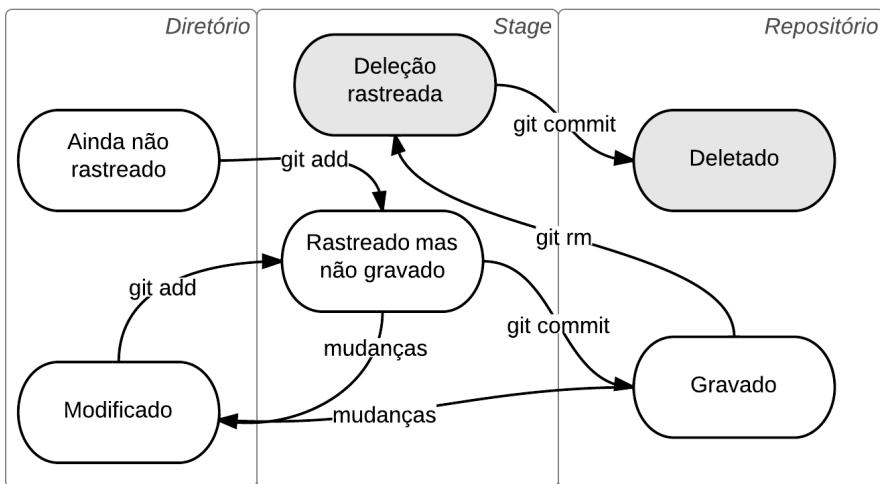



Figura 3.3: Removendo arquivos com Git

> Um detalhe importante é saber que, apesar de o arquivo ter sido removido, seu conteúdo fica gravado no histórico do repositório. Dessa maneira, é possível obtermos qualquer arquivo que já existiu em nosso repositório.
>
> Porém, se por descuido comitarmos algum arquivo grande e depois deletá-lo, nosso repositório não diminuirá de tamanho, já que o histórico será mantido. Mais adiante, veremos como limpar nosso repositório, expurgando arquivos grandes, utilizando ferramentas apropriadas.

3.7 Renomeando e movendo arquivos

Renomeando arquivos

Para manter o padrão de nomenclatura, resolvemos modificar o nome do arquivo `estilos.css` para `principal.css`.

Para fazer isso, teríamos que criar o novo arquivo, copiar seu conteúdo, remover o arquivo antigo, adicionando tanto o novo arquivo como a deleção do arquivo antigo na área de *stage*.

Bastante trabalho, não? Ainda bem que esse trabalho todo pode ser poupado com o comando:

```
$ git mv estilos.css principal.css
```

Se executarmos o comando `git status`, teremos:

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    renamed:    estilos.css -> principal.css
#
```

Podemos observar que o arquivo aparece como renomeado na área de stage e pronto para ser comitado. E, claro, se observarmos no diretório do nosso repositório, o arquivo foi realmente renomeado.

É interessante saber que, se tivéssemos renomeado o arquivo da maneira mais trabalhosa, o Git iria detectar o nosso objetivo, depois que tivéssemos efetuados todos os comandos necessários. Isso acontece porque o Git rastreia o conteúdo dos arquivos, não apenas o nome.

Antes de continuarmos, precisamos atualizar o nome do arquivo CSS em `index.html`:

```
<!-- início do arquivo ... -->
    <head>
        <!-- tags meta ... -->
        <link rel="stylesheet" href="principal.css">
        <!-- tags title e script ... -->
    </head>
<!-- restante do arquivo ... -->
```

Feitas as alterações, podemos gravá-las no repositório com o comando `git commit -am "Renomeando CSS"`. Na saída, deve aparecer:

```
[master 27ffd5e] Renomeando CSS
 2 files changed, 1 insertion(+), 1 deletion(-)
 rename estilos.css => principal.css (100%)
```

Pronto! Arquivo renomeado!

Movendo arquivos

E se quisermos mover o arquivo `principal.js` para um subdiretório chamado `js`?

Primeiramente, devemos criar o diretório `js`.

Depois, deveríamos criar um arquivo `principal.js` dentro desse novo diretório, copiar os conteúdos do arquivo anterior e adicionar o novo diretório à área de stage. Feito isso, precisaríamos remover o arquivo atual, adicionando a remoção à stage. Só então poderíamos comitar as alterações.

Todo esse trabalho também pode ser poupado pelo comando `git mv`:

```
$ git mv principal.js js/principal.js
```

Atenção: é importante que o diretório `js` já tenha sido criado!

Se executarmos o comando `git status`, teremos:

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    renamed:    principal.js -> js/principal.js
#
```

Note que o arquivo `principal.js` foi descrito como renomeado para o novo arquivo, que está dentro do subdiretório `js`. Para o Git, não há diferença entre um arquivo movido ou renomeado. No fim das contas, aquele conteúdo (o que é efetivamente rastreado) mudou de local.

Vamos atualizar o arquivo `index.html` para apontar para a localização correta do arquivo `principal.js`:

```
<!-- início do arquivo ... -->
    <head>
        <!-- tags meta, link e title ... -->
        <script src="js/principal.js"></script>
    </head>
<!-- restante do arquivo ... -->
```

Então, comitar as alterações com `git commit -am "Movendo principal.js"`. Na saída, deve aparecer:

```
[master 3dd877f] Movendo principal.js
 2 files changed, 1 insertion(+), 1 deletion(-)
 rename principal.js => js/principal.js (100%)
```

Pronto! Arquivo movido!

Incluindo a movimentação de arquivos na nossa visualização, ficaríamos com:

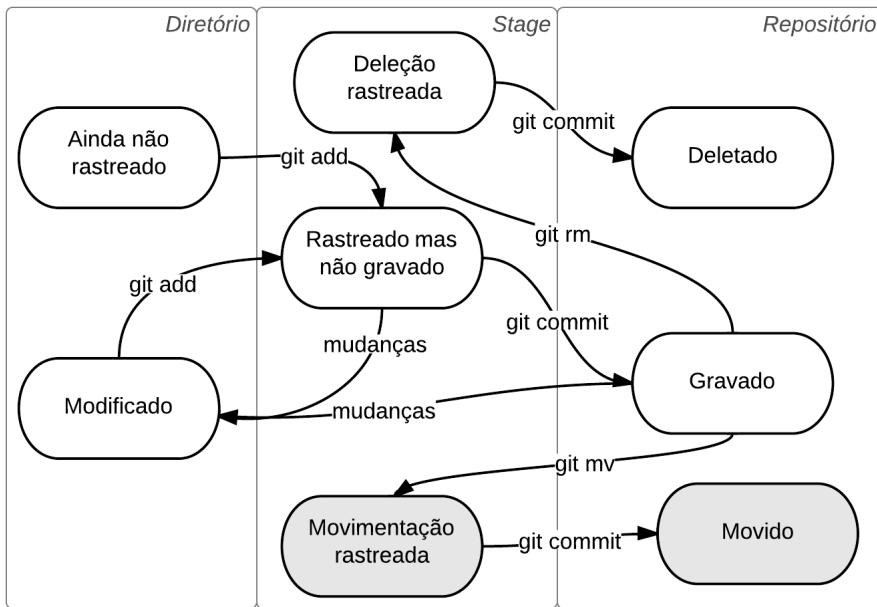



Figura 3.4: Renomeando e movendo arquivos com Git

3.8 Desfazendo mudanças

Desfazendo mudanças não rastreadas

Vamos dizer que o dono da empresa Móveis S. A. ligou de madrugada depois de tomar umas e outras, pedindo para colocarmos na página principal, um texto um tanto peculiar.

Para atendê-lo, editamos o arquivo `index.html`, colocando o seguinte conteúdo no fim do arquivo:

```
<!-- inicio do arquivo -->
        <p>Móveis baratos pra c*&%$#@!</p>
    </body>
</html>
```

Fizemos a alteração sem muita convicção. Por isso, não a adicionamos na área de stage. A resposta de um `git status` seria:

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working
directory)
#
#    modified:   index.html
#
no changes added to commit (use "git add" and/or
"git commit -a")
```

Logo de manhã, o dono da empresa liga arrependido, pedindo para desfazermos as alterações.

Como podemos fazer de maneira fácil?

Devemos executar o seguinte comando:

```
$ git checkout -- index.html
```

O comando `git checkout` desfaz as alterações ainda não rastreadas, ou seja, que ainda não estão na área de stage, voltando ao conteúdo anterior do arquivo.

Após um `git status` teríamos:

```
# On branch master
nothing to commit, working directory clean
```

Ao verificarmos o conteúdo do arquivo `index.html`, podemos constatar que a nossa mudança foi desfeita e o arquivo ficou com o conteúdo original, conforme o último commit.

> Caso haja alguma mudança já rastreada no arquivo, dentro da área de stage, ao executarmos o comando `git checkout`, apenas as alterações indesejadas, fora da stage, serão desfeitas. As mudanças que já estavam na stage permanecerão. Serão desfeitas apenas as alterações, que ainda não tinham sido rastreadas.

E se apagarmos algum arquivo sem querer? Medo! Desespero!

Vamos dizer que apagamos o arquivo `index.html`.

Ao executarmos o comando `git status`, teríamos:

```
# On branch master
# Changes not staged for commit:
#   (use "git add/rm <file>..."to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working
directory)
#
#    deleted:    index.html
#
no changes added to commit (use "git add" and/or
"git commit -a")
```

Quem poderá nos ajudar? O Git, é claro.

Podemos utilizar o comando `git checkout` para recuperar arquivos removidos acidentalmente:

```
$ git checkout -- index.html
```

Pronto! Arquivo recuperado!

Desfazendo mudanças já rastreadas

E se já tivermos rastreado uma mudança indesejada no arquivo? Como voltar atrás?

Vamos dizer que fizemos a seguinte modificação no fim do arquivo `index.html`:

```
<!-- inicio do arquivo -->
        <p>Móveis baratos pra p&%$#!</p>
    </body>
</html>
```

Logo depois, colocamos a mudança na área de stage com o comando `git add index.html`. Ao executarmos o `git status`, teríamos:

```
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#    modified:   index.html
#
```

Se quisermos apenas remover da área de stage a mudança efetuada no arquivo `index.html`, preservando o arquivo modificado, devemos executar:

```
$ git reset -- index.html
```

Quando utilizado dessa maneira, apenas informando um arquivo que tem mudanças na área de stage, o comando `git reset` retira o arquivo da stage mas preserva tudo o que foi modificado nesse arquivo.

Ao verificarmos o arquivo `index.html`, veremos que as alterações que fizemos anteriormente estarão intocadas.

Depois de executarmos o comando `git status`, veremos que as mudanças aparecem fora da área de stage (*Changes not staged for commit*):

```
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
# (use "git checkout -- <file>..." to discard changes in working
directory)
#
#    modified:   index.html
#
no changes added to commit (use "git add" and/or
"git commit -a")
```

> Se invocarmos o comando `git reset` sem nenhum parâmetro, serão retirados todos os arquivos da área de stage. As alterações efetuadas nesses arquivos serão preservadas.

No caso de querermos descartar todas as mudanças nos arquivos ao invocarmos `git reset`, devemos utilizar a opção `--hard`.

Há um detalhe importante: a opção `--hard` retira todos os arquivos da área de stage e desfaz todas as alterações nesses arquivos. No fim das contas, o repositório fica exatamente no estado que estava no último commit.

Para testarmos, vamos colocar as mudanças no arquivo `index.html` novamente na área de stage executando `git add index.html`.

Então, vamos executar o comando:

```
$ git reset --hard
```

Deve aparecer algo como:

```
HEAD is now at 3dd877f Movendo principal.js
```

Todos os arquivos foram retirados da área de stage e todas as alterações nesses arquivos foram desfeitas. Podemos confirmar isso executando o comando `git status` novamente:

```
# On branch master
nothing to commit, working directory clean
```

Ou seja, depois do `git reset --hard`, o repositório voltou a ficar exatamente como estava no último commit. Nenhum arquivo estará modificado e a área de stage estará vazia.

Desfazendo mudanças já comitadas

Agora, e se já tivermos comitado algumas modificações e quisermos voltar atrás?

Por exemplo, se tivermos feito as seguintes mudanças no arquivo `index.html`:

```
<!-- inicio do arquivo -->
        <p>Móveis baratos pra b&%$#!</p>
    </body>
</html>
```

Depois, comitamos as alterações:

```
$ git commit -am "Adicionando texto peculiar"
```

Na saída, teríamos algo como:

```
[master 172691b] Adicionando texto peculiar
 1 file changed, 1 insertion(+)
```

Se quisermos voltar atrás, desfazendo as alterações no repositório, podemos utilizar o comando:

```
$ git revert --no-edit 172691b
```

Nesse comando, o código `172691b` representa o último commit efetuado. Se omitirmos a opção `--no-edit`, será aberto um editor de texto para editarmos a mensagem do novo commit.

> Em vez de passar o código do último commit como parâmetro para o `git revert`, poderíamos ter utilizado `HEAD` que, no nosso caso, aponta para o último commit.

Após executar o `git revert`, teríamos como resposta algo parecido com:

```
[master 9274519] Revert "Adicionando texto peculiar"
 1 file changed, 1 deletion(-)
```

Ao verificarmos o arquivo `index.html` veremos que as alterações do último commit foram desfeitas, retornando à versão anterior.

Se observarmos novamente o histórico de commits do nosso repositório, com o comando `git log -n 2 --oneline`, teremos:

```
9274519 Revert "Adicionando texto peculiar"
172691b Adicionando texto peculiar
```

Note que o comando `git revert` efetuou um novo commit com a versão anterior dos arquivos. Foi utilizada a mensagem *Revert* seguida da mensagem do commit anterior (no exemplo, *Revert "Adicionando texto peculiar"*).

> No caso de passarmos um código de commit antigo, apenas as alterações feitas naquele commit serão desfeitas. Isso é algo bastante poderoso!
>
> Imagine que você descubra que um *bug* veio de uma alteração de determinado commit em um conjunto de arquivos. Apenas com um `git revert` é possível desfazer as alterações que inseriram o *bug*.
>
> Por isso, vale reiterar que commits pequenos e bem montados são importantes para podermos utilizar o real poder do Git.

Uma outra forma de desfazer alterações já comitadas é utilizando o comando `git reset` seguido da opção `--hard` e de um código de commit.

Se quisermos voltar ao commit anterior às alterações peculiares solicitadas pelo nosso cliente, devemos executar:

```
$ git reset --hard 3dd877f
```

Teríamos como resposta algo similar a:

```
HEAD is now at 3dd877f Movendo principal.js
```

Se observamos os arquivos, as últimas alterações foram desfeitas.

No histórico do repositório, exibido ao executarmos `git log --oneline`, os commits descartados não aparecem:

```
3dd877f Movendo principal.js
27ffd5e Renomeando CSS
3234100 Removendo página de produtos
65727bb Página de produtos
8aa07bd Diminuindo intervalo de troca de banner
2d04ca8 Banner ao abrir a pagina
842d2cf Inserindo arquivo principal.js
bb229c6 Script de troca de banner
ccff92c Inserindo titulo e diminuindo tamanho da pagina
4a4fdf7 Commit inicial
```

Ao utilizarmos o comando `git reset` da maneira anterior, com a opção `--hard` e um código de commit, o histórico do repositório é reescrito.

Porém, é importante mantermos todas as alterações gravadas no repositório, mesmo que indesejadas. Afinal de contas, estamos utilizando um sistema de controle de versões justamente para ter um histórico fiel da evolução do código de nossos projetos.

Por isso, nas raras ocasiões em que é necessário desfazer commits, **recomendamos que seja utilizado o comando `git revert`**.

Atualizando nossa visualização gráfica do que aprendemos, agora com comandos para desfazer modificações, temos:

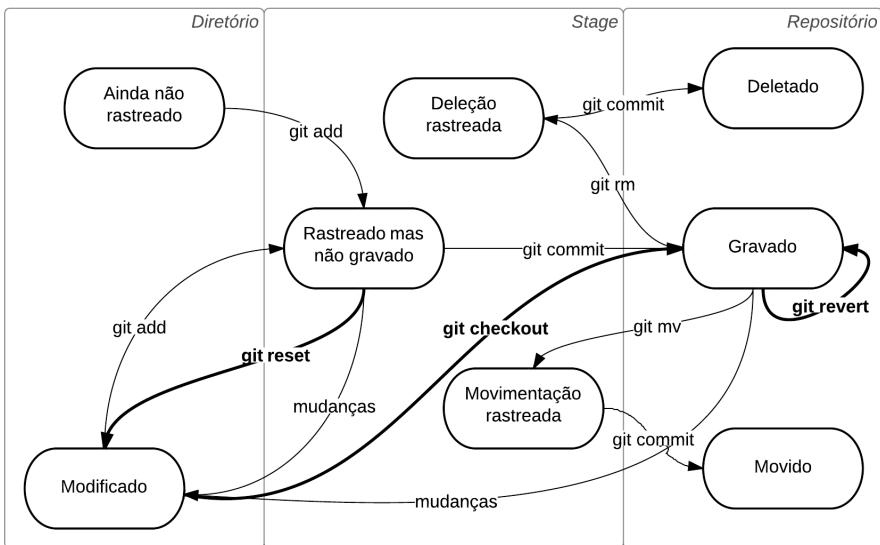



Figura 3.5: Desfazendo mudanças com Git

Capítulo 4

Trabalhando com repositório remoto

Nosso projeto já possui vários arquivos, e também alguns commits que fizemos para guardar o histórico das modificações realizadas. Entretanto, todos os commits estão registrados apenas localmente, ou seja, unicamente no nosso computador.

Se algum dia nosso computador apresentar algum problema técnico, e deixar de funcionar, ou até mesmo se ele for roubado, teríamos um sério problema, pois junto com ele perderíamos também nosso repositório e o projeto.

Podemos adotar uma estratégia de realizar *backups* semanalmente do nosso repositório, ou até mesmo diariamente, para que assim sempre tenhamos uma cópia de segurança.

Entretanto, essa estratégia de realização de backups é desaconselhada por

ser trabalhosa e sujeita a falhas, já que podemos nos esquecer de realizar o backup em algum dia.

Além disso, manter o repositório apenas localmente nos traz outra questão: como fazer para trabalhar em equipe, com vários desenvolvedores alterando os arquivos do projeto e registrando essas alterações com commits?

Certamente ficar compactando e enviando o repositório para os outros membros da equipe, a cada novo commit, não parece ser uma ideia muito boa.

4.1 Repositório remoto

Para evitar os problemas citados anteriormente devemos trabalhar com um **repositório remoto**, que nada mais é do que um *repositório Git* criado em outro computador, chamado de **servidor**.

Se os desenvolvedores da equipe trabalham na mesma empresa, todos eles vão estar conectados pela rede. Sendo assim, para criarmos um repositório remoto, devemos primeiramente utilizar algum computador da empresa que esteja conectado na rede, e criar o repositório Git do projeto neste computador, que passará a ser o servidor central do projeto.

Criando um repositório remoto

A criação do repositório remoto é feita com o comando `git init`, da mesma maneira que fizemos para criar o repositório local, entretanto devemos passar o parâmetro `--bare` ao comando:

```
$ git init --bare moveis-ecologicos.git
```

Ao executar o comando, será apresentada uma mensagem como:

```
Initialized empty Git repository
in /opt/repositorios/moveis-ecologicos.git/
```

O parâmetro `--bare` serve para que o Git não crie um *working tree* (diretório de trabalho), impedindo que commits sejam efetuados diretamente no servidor. No nosso caso faz sentido, já que os commits serão realizados pelos

desenvolvedores, localmente, em seus computadores, e depois esses commits serão enviados e armazenados no repositório remoto, localizado no servidor.

Ao executar o comando `git init --bare moveis-ecologicos.git`, o Git criará um novo diretório chamado `moveis-ecologicos.git` com a seguinte estrutura:

Figura 4.1: Estrutura de diretórios do repositório Git

Lembre-se de **nunca** mexer nesses arquivos e diretórios, pois eles representam o repositório Git e devem ser manipulados apenas pelo Git.

4.2 Adicionando o repositório remoto

Agora que o repositório remoto já foi criado no servidor, é possível enviar os commits efetuados no nosso repositório local para o repositório remoto. Mas antes disso precisamos, de alguma maneira, indicar ao Git onde está localizado o repositório remoto.

Para ensinar ao Git onde se encontra o nosso repositório remoto, devemos utilizar o comando `git remote add`, no qual informamos o **endereço** do repositório remoto.

Supondo que o nosso servidor esteja na rede com o endereço IP `192.168.10.99`, e o repositório remoto do nosso projeto tenha sido criado no diretório `/opt/repositorios/moveis-ecologicos.git`, para adicionarmos o repositório remoto devemos executar o comando `git remote add` da seguinte maneira:

```
$ git remote add servidor
file://192.168.10.99/opt/repositorios/moveis-ecologicos.git
```

Ao executar o comando `git remote add` devemos informar o `name` (nome) do repositório remoto e sua `url` (endereço).

No exemplo anterior, o `name` do repositório adicionado foi **servidor** e a `url` foi **file://192.168.10.99/opt/repositorios/moveis-ecologicos.git**.

É possível adicionar mais de um repositório remoto, desde que cada um tenha seu `name` distinto. Isto pode ser útil se precisarmos enviar os commits para mais de um servidor.

Listando os repositórios remotos

Para listar os repositórios remotos que foram adicionados devemos utilizar o comando `git remote`:

```
$ git remote
servidor
```

Mas repare que é exibido apenas o `name` dos repositórios remotos. Para que também seja exibida a `url` devemos adicionar o parâmetro `-v` ao comando:

```
$ git remote -v
servidor file://192.168.10.99/opt/repositorios/
moveis-ecologicos.git (fetch)
servidor file://192.168.10.99/opt/repositorios/
moveis-ecologicos.git (push)
```

Observe que o repositório remoto foi listado duas vezes. Isso acontece pois o Git permite que tenhamos duas `URLs` distintas para o mesmo repositório remoto, sendo uma para leitura (*fetch*) e outra para escrita (*push*). Isto pode ser útil se precisarmos utilizar protocolos distintos para leitura e escrita.

Alterando e removendo os repositórios remotos

É possível alterar o `name` de um repositório remoto utilizando o comando `git remote rename`:

```
$ git remote rename servidor outronome
$ git remote
outronome
```

O que aconteceria se alguém alterasse o endereço IP do servidor de `192.168.10.99` para `192.168.10.110`? Teríamos um problema, pois nosso repositório remoto estaria apontando para o endereço antigo. Devemos nesse caso alterar a `url` do repositório remoto, e isto é feito com outro comando, o `git remote set-url`, passando como parâmetro o `name` do repositório remoto e a nova `url`:

```
$ git remote -v
    servidor file://192.168.10.99/opt/repositorios/
    moveis-ecologicos.git (fetch)
    servidor file://192.168.10.99/opt/repositorios/
    moveis-ecologicos.git (push)

    $ git remote set-url servidor
file://192.168.10.110/opt/repositorios/moveis-ecologicos.git

    $ git remote -v
    servidor file://192.168.10.110/opt/repositorios/
    moveis-ecologicos.git (fetch)
    servidor file://192.168.10.110/opt/repositorios/
    moveis-ecologicos.git (push)
```

4.3 Enviando commits para o repositório remoto

Agora que já adicionamos o repositório remoto, podemos, enfim, enviar os commits para o servidor.

Para enviar os commits locais, que ainda não existem no servidor, devemos utilizar o comando `git push`, informando o `name` do repositório remoto, no nosso caso **servidor**, seguido de **master**:

```
$ git push servidor master
```

Será exibida uma mensagem como:

```
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 227 bytes | 0 bytes/s, done.
Total 2 (delta 1), reused 0 (delta 0)
To file://192.168.10.99/opt/repositorios/moveis-ecologicos.git/
    e573bbb..8a2481d  master -> master
```

Pronto! Agora os commits já foram enviados para o servidor.

> A palavra `master` que utilizamos no comando anterior é o nome da *branch* principal do nosso repositório. Estudaremos branches com profundidade mais adiante.

4.4 Clonando o repositório remoto

Nosso projeto continua a crescer e não param de chegar mais pedidos de funcionalidades. Tivemos que contratar mais um desenvolvedor para a equipe, e ele já está pronto para começar a desenvolver as novas funcionalidades do sistema, mas para isso vai precisar, primeiramente, obter uma cópia do repositório com todos os arquivos e commits.

Para obtermos uma cópia de um repositório Git, devemos utilizar o comando `git clone` informando a URL do repositório a ser clonado:

```
$ git clone file://192.168.10.99/opt/repositorios/
moveis-ecologicos.git
```

Será exibida uma mensagem como:

```
Cloning into 'projeto'...
remote: Counting objects: 8, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 8 (delta 1), reused 0 (delta 0)
Receiving objects: 100% (8/8), 618.32 KiB | 0 bytes/s, done.
Resolving deltas: 100% (1/1), done.
Checking connectivity... done.
```

Ao executar o comando anterior, o Git criará um diretório chamado `moveis-ecologicos`, que será uma cópia local do repositório remoto, contendo todos os arquivos e o histórico dos commits realizados.

Por padrão, o Git criará um diretório com o mesmo nome do repositório, e também já adicionará um repositório remoto com o name `origin`, que aponta para a URL clonada.

4.5 Sincronizando o repositório local

Após clonar o repositório do projeto, o novo desenvolvedor finalizou uma das novas funcionalidades, realizando alguns commits para isso, e inclusive já enviou tais commits para o servidor.

Sendo assim, para que possamos visualizar as alterações e commits realizados pelo novo desenvolvedor, devemos **sincronizar** o nosso repositório local com o servidor, **puxando** os novos commits para o nosso repositório local.

A sincronização do repositório local com o servidor é feita com o uso do comando `git pull`, onde devemos informar o `name` do repositório remoto que queremos sincronizar, e o `nome` da branch local a ser atualizada:

```
$ git pull servidor master
```

Será exibida uma mensagem como:

```
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 2 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (2/2), done.
From file://192.168.10.99/opt/repositorios/moveis-ecologicos
 * branch            master     -> FETCH_HEAD
   8a2481d..61131f4  master     -> servidor/master
Updating 8a2481d..61131f4
Fast-forward
 promocoes.html | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 promocoes.html
```

Agora nosso repositório local está atualizado, e já é possível visualizar as alterações e commits efetuados pelo novo desenvolvedor.

Repositório centralizado ou distribuído

O Git é um sistema de controle de versão distribuído, e por isso não depende de um servidor central, diferentemente do CVS e SVN. Conforme visto no capítulo anterior, podemos trabalhar apenas com o repositório local.

Entretanto, conforme visto neste capítulo, é possível trabalhar com o Git de maneira centralizada, onde um servidor será o repositório central, recebendo os commits de todos os desenvolvedores, e estes, por sua vez, sincronizando seus repositórios locais com o servidor, frequentemente, para obter os novos commits. Esta é a maneira mais comum de utilização do Git.

4.6 Protocolos suportados pelo Git

Quando adicionamos ou clonamos um repositório Git, devemos informar a `url` do repositório, que utiliza algum protocolo para comunicação e transferência de dados.

O Git suporta quatro protocolos para comunicação e transferência de dados:

- Local
- SSH
- Git
- HTTP/HTTPS

Protocolo local

O protocolo local foi o utilizado neste capítulo. Ele pode ser utilizado quando o repositório remoto estiver localizado no mesmo computador em

que se encontra o repositório local, ou em outro computador que esteja conectado na mesma rede.

A utilização do protocolo local é feita com o uso do prefixo `file://` na URL do repositório a ser clonado:

```
$ git clone file:///opt/repositorios/moveis-ecologicos.git
```

É possível omitir o prefixo `file://` na URL, bastando apenas informar o caminho do repositório:

```
$ git clone /opt/repositorios/moveis-ecologicos.git
```

As vantagens da utilização do protocolo local são a sua simplicidade, e o uso das permissões existentes de acesso a arquivos e diretórios. Já uma das desvantagens é a dificuldade de configuração para acesso externo à rede.

Protocolo SSH

O protocolo SSH é, provavelmente, o mais utilizado, por ser rápido, seguro, simples de configurar e por suportar tanto a leitura quanto a escrita de dados.

O uso do protocolo SSH é feito com a URL seguindo o padrão `usuario@servidor:/caminho/repositorio.git`. Por exemplo:

```
$ git clone root@192.168.10.99:/opt/repositorios/
moveis-ecologicos.git
```

Protocolo Git

O Git possui um protocolo próprio, que é similar ao SSH, mas sem o mecanismo de autenticação. Por conta disso, ele acaba sendo mais rápido; entretanto não é seguro, e seu uso é apenas para leitura.

Para clonar um repositório utilizando o protocolo Git, a URL deve possuir o prefixo `git://`:

```
$ git clone git://192.168.10.99/opt/repositorios/
moveis-ecologicos.git
```

Protocolo HTTP/HTTPS

O Git também suporta o protocolo HTTP, que é bastante utilizado quando estamos trabalhando em empresas que possuem um controle rígido de segurança, e a porta 22, utilizada pelo protocolo SSH, é bloqueada.

Para clonar um repositório utilizando o protocolo HTTP, a URL deve possuir o prefixo `http://`:

```
$ git clone http://192.168.10.99/opt/repositorios/
moveis-ecologicos.git
```

Também é possível utilizar o protocolo HTTPS, que adiciona uma camada de segurança sobre o HTTP, com a utilização do protocolo SSL/TLS:

```
$ git clone https://192.168.10.99/opt/repositorios/
moveis-ecologicos.git
```

Capítulo 5

Hospedando o repositório no GitHub

No capítulo anterior, vimos como utilizar um repositório remoto, para que tenhamos facilidade ao trabalhar em equipe no projeto. Agora, sempre que um novo membro da equipe for trabalhar no projeto, ele deve primeiramente clonar o repositório, efetuar as alterações necessárias, comitar tais alterações, e então enviar os commits para o repositório remoto.

Assim sendo, vimos que a utilização de um repositório remoto nos ajuda a ter uma melhor organização para trabalhar em equipe em um projeto, além de também funcionar como backup.

Entretanto, como o repositório remoto está localizado em um servidor da empresa, ficamos limitados a efetuar a sincronização apenas dentro dela, pois precisaremos estar conectados à rede local para conseguirmos acessar o

servidor.

Se algum dia um desenvolvedor precisar sincronizar as alterações com o repositório remoto de fora da empresa, por exemplo de casa, isso não vai ser possível, pois ele não vai estar conectado à rede local.

Até é possível acessar a rede local da empresa externamente, mas para isso será necessário configurar uma VPN (*Virtual Private Network*), ou liberar acesso externo ao servidor via SSH, o que pode ser algo complicado de se fazer, além de também poder ser proibido por algumas empresas, devido a políticas de segurança.

O ideal seria que o repositório remoto estivesse acessível pela internet, pois assim poderíamos acessá-lo de qualquer lugar do mundo, bastando apenas estarmos conectados à internet.

5.1 Serviços de hospedagem de projetos

Existem alguns serviços para hospedagem de projetos na internet, que permitem a utilização de ferramentas de controle de versão. A ideia é que você possa hospedar seus projetos, juntamente com os repositórios, na internet, e assim será possível acessá-los de qualquer lugar do mundo.

Dentre os principais serviços que suportam o Git, estão:

- GitHub
- Bitbucket
- Google Code

5.2 GitHub: a rede social dos desenvolvedores

Criado em 2008 por Tom Preston-Werner, Chris Wanstrath e PJ Hyett, com o objetivo de simplificar o compartilhamento de projetos, o GitHub é hoje o maior serviço de hospedagem de repositórios de projetos, contendo, atualmente, mais de 3 milhões de usuários e mais de 12 milhões de repositórios.

Figura 5.1: Logotipo do GitHub

É o principal serviço utilizado pelos desenvolvedores que utilizam o Git. Vários projetos importantes estão hospedados no GitHub, dentre eles:

- jQuery
- Ruby on Rails
- Node.js
- Django
- Bootstrap

O GitHub é também uma ferramenta colaborativa, pois nos permite interagir nos repositórios, por meio de algumas funcionalidades como:

- Issue Tracker: para criação e gestão de bugs e milestones do projeto;
- Pull Requests: para que outros usuários possam enviar seus commits com alterações no projeto, ou commits com correções de bugs;
- Commit Comments: para que os usuários possam comentar e discutir sobre as modificações no código, de um determinado commit.

Além disso, no GitHub é possível criar **organizações**, e dentro delas criar **equipes**, para que então seja possível vincular determinados usuários a determinadas equipes, bem como vincular as equipes aos repositórios em que elas

vão trabalhar. Isso é muito útil para empresas que possuem muitos projetos e muitos colaboradores, pois permite uma melhor organização.

Por conta dessas funcionalidades e por permitir uma maior colaboração nos repositórios, o GitHub é considerado por muitos como uma espécie de **rede social** para desenvolvedores.

5.3 Encontrando projetos e visualizando o código-fonte

Vamos acessar o site do GitHub e visualizar o código-fonte de algum projeto open source. Para isso, basta abrir um navegador e acessar o endereço: http://github.com

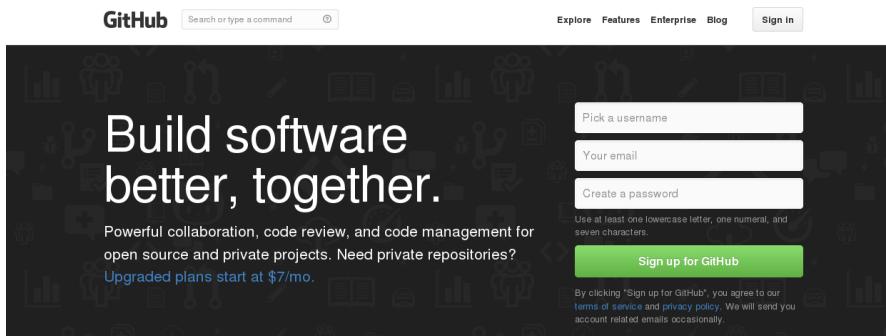



Figura 5.2: Página inicial do GitHub

Essa é a página inicial do GitHub, nela podemos efetuar login, cadastrar um novo usuário, fazer buscas e acessar alguns links interessantes.

No topo da página existe um campo de texto, usado para pesquisar por repositórios ou usuários. Vamos pesquisar pelo repositório do **jQuery**, uma famosa biblioteca de JavaScript, bastando para isso digitar a palavra **jquery** no campo de texto e apertar a tecla `Enter`.

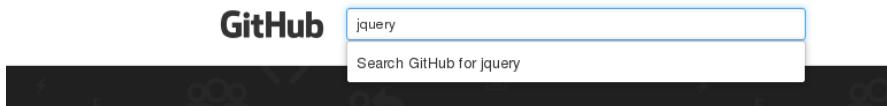



Figura 5.3: Campo para pesquisar por repositórios

Após realizar a pesquisa, será apresentada uma página com o resultado, listando todos os repositórios que possuam no nome ou na descrição a palavra **jquery**.

Figura 5.4: Página com o resultado da pesquisa

Vamos selecionar o primeiro resultado, que no caso é o projeto **jQuery** que estamos buscando.

Figura 5.5: Link para o repositório do jQuery

Na página do repositório do jQuery é possível navegar pelo código-fonte, visualizar a descrição do projeto, os commits, os usuários que contribuem para o projeto, e até mesmo baixar o repositório compactado em um arquivo `zip`.

Figura 5.6: Página do repositório do jQuery

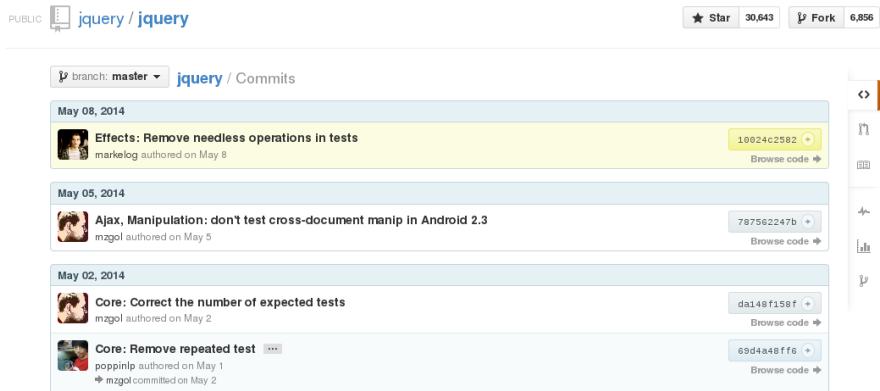
Visualizando os commits do projeto

Vamos agora visualizar os últimos commits realizados no projeto. Para isso, basta clicar no link **commits**:

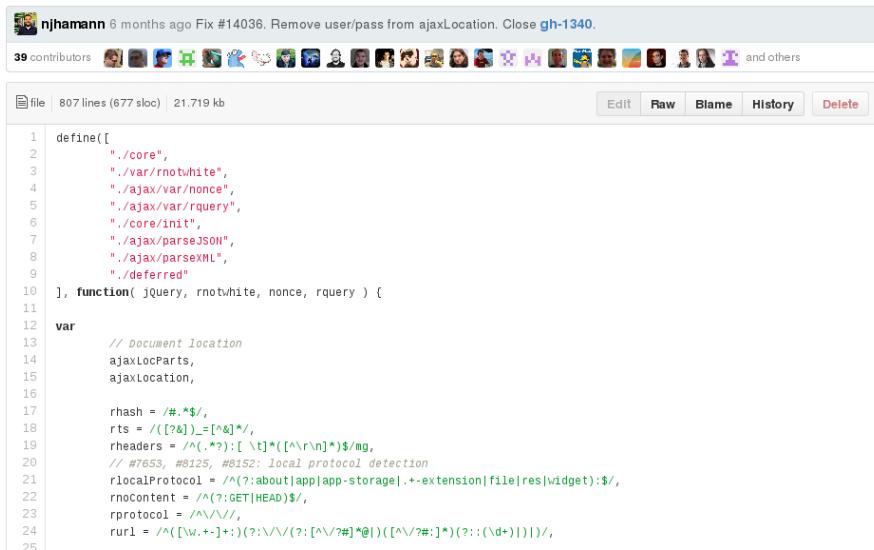
Figura 5.7: Link para a página de commits do jQuery

Será apresentada uma página listando os últimos commits efetuados no

projeto, onde também é possível navegar por commits mais antigos. Visualizar esta página seria como executar o comando `git log` no repositório.

Figura 5.8: Página de commits do jQuery

Ao clicar em algum dos commits, será apresentada uma página de detalhamento, onde podemos visualizar as alterações efetuadas, similar a quando executamos o comando `git diff`:

Figura 5.9: Página de detalhamento de commit

Visualizando o código-fonte do projeto

De volta à página principal do repositório, agora vamos navegar pelo código-fonte. Já na página principal do repositório é possível visualizar os arquivos e diretórios do projeto. Vamos navegar pelo diretório **src**:

Figura 5.10: Diretórios do repositório

Serão exibidos todos os arquivos e subdiretórios existentes, e podemos detalhar algum deles, por exemplo o arquivo `ajax.js`:

njhamann 6 months ago Fix #14036. Remove user/pass from ajaxLocation. Close gh-1340.

39 contributors and others

file | 807 lines (677 sloc) | 21.719 kb — Edit Raw Blame History Delete

```
1  define([
2  	"./core",
3  	"./var/rnotwhite",
4  	"./ajax/var/nonce",
5  	"./ajax/var/rquery",
6  	"./core/init",
7  	"./ajax/parseJSON",
8  	"./ajax/parseXML",
9  	"./deferred"
10 ], function( jQuery, rnotwhite, nonce, rquery ) {
11
12 var
13 	// Document location
14 	ajaxLocParts,
15 	ajaxLocation,
16
17 	rhash = /#.*$/,
18 	rts = /([?&])_=[^&]*/,
19 	rheaders = /^(.*?):[ \t]*([^\r\n]*)$/mg,
20 	// #7653, #8125, #8152: local protocol detection
21 	rlocalProtocol = /^(?:about|app|app-storage|.+-extension|file|res|widget):$/,
22 	rnoContent = /^(?:GET|HEAD)$/,
23 	rprotocol = /^\/\//,
24 	rurl = /^([\w.+-]+:)(?:\/\/(?:[^\/?#]*@|)([^\/?#:]*)(?::(\d+)|)|)/,
25
```

Figura 5.11: Detalhamento de um dos arquivos do repositório

Além de ser exibido todo o código-fonte do arquivo, nessa página também é possível visualizar todos os usuários que já efetuaram commits com modificações neste arquivo.

5.4 Criando um usuário no GitHub

Para poder começar a usar o GitHub e criar os nossos repositórios, devemos primeiramente cadastrar um novo usuário no site.

Atenção! Caso você já tenha criado um usuário anteriormente, no capítulo 2 você pode seguir para a próxima seção.

A criação de um novo usuário no GitHub deve ser feita na página inicial do site: http://github.com

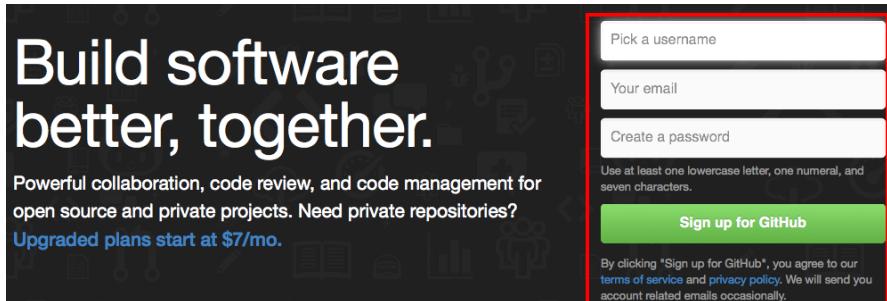



Figura 5.12: Página de criação de novo usuário

Devemos informar um nome de usuário, o e-mail e uma senha. Caso já exista um outro usuário cadastrado com o mesmo nome de usuário e/ou e-mail informado, o GitHub nos mostrará um alerta indicando tal situação, e só permitirá que continuemos o cadastro após a alteração dessa(s) informação(ões).

Após informar os dados e clicar no botão **Sign up for GitHub**, seremos redirecionados para uma página de boas-vindas, onde devemos escolher qual o plano a ser contratado.

Figura 5.13: Página de boas-vindas e escolha do plano de contratação

Planos de contratação oferecidos pelo GitHub

O GitHub possui alguns planos que podemos contratar, sendo um gratuito, e os outros com custos mensais.

No plano gratuito do GitHub, o usuário não tem nenhum custo, entretanto ele somente poderá criar repositórios públicos, ou seja, qualquer pessoa poderá encontrar seus repositórios e inclusive ter acesso ao código-fonte. Este plano é ideal para usuários que querem criar e/ou colaborar com projetos open source.

Mas muitas empresas possuem projetos que são restritos, e não podem ter o código-fonte acessível a qualquer pessoa. Neste caso, a empresa deverá contratar algum dos planos pagos oferecidos pelo GitHub.

A diferença entre os planos pagos está na quantidade de repositórios pri-

vados que poderão ser criados. No plano mais barato, que atualmente custa $7 por mês, é possível criar até 5 repositórios privados, enquanto que no plano mais caro, que atualmente custa $50 por mês, essa quantidade de repositórios privados sobe para 50.

Após escolher o plano que mais se adeque às nossas necessidades, devemos finalizar o cadastro clicando no botão **Finish sign up**, e seremos redirecionados para a página que mostra os repositórios do nosso usuário e outras informações. Essa página é conhecida como **Dashboard**.

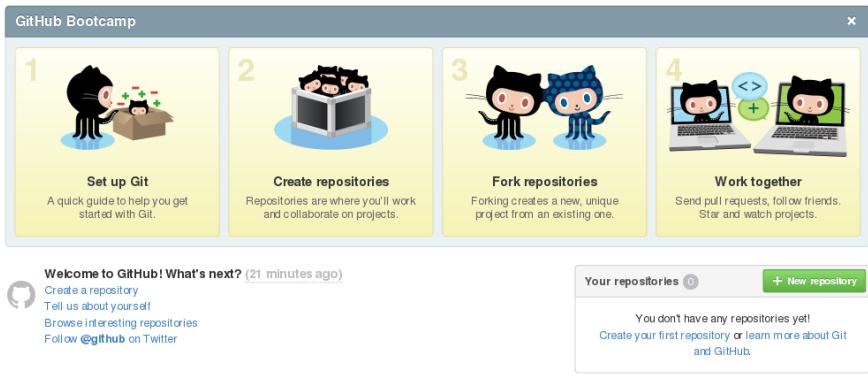



Figura 5.14: Página de dashboard

O GitHub enviará um e-mail de confirmação de cadastro, com um link para validação e ativação. É importante acessarmos esse link, pois assim o e-mail será validado e associado ao nosso usuário, e além disso, todas as notificações do GitHub serão enviadas apenas ao(s) e-mail(s) cadastrado(s).

5.5 Criando o repositório do projeto

Agora que temos um usuário cadastrado, já é possível criar o repositório do projeto.

Para criar um novo repositório no GitHub, devemos acessar a URL http://github.com/new ou clicar no botão **New repository** existente na página principal.

Figura 5.15: Página de criação de novo repositório

Nesta página devemos preencher o nome do repositório no campo `Repository name`, e, opcionalmente, preencher uma descrição no campo `Description`. Após isso, basta clicar no botão `Create repository`, e seremos redirecionados para a página principal do repositório, onde serão exibidas as informações de como adicionar o repositório remoto, para podermos sincronizar os commits do nosso repositório local.

Figura 5.16: Página do novo repositório

Agora basta adicionarmos o repositório remoto, com o comando `git remote add`, conforme exibido na página do repositório:

```
$ git remote add origin
https://github.com/fulanodasilva/moveis-ecologicos.git
```

5.6 Enviando os commits do projeto para o GitHub

Com o repositório remoto adicionado, podemos finalmente enviar os commits para o GitHub.

O envio dos commits é feito da mesma maneira que vimos no capítulo anterior, ou seja, utilizando o comando `git push`:

```
$ git push origin master
```

Como utilizamos o protocolo `https` ao adicionar o repositório remoto, sempre que formos sincronizar nosso repositório local, o Git nos solicitará o usuário e senha cadastrados no GitHub.

Após o Git enviar os commits para o repositório remoto, será exibida uma mensagem como:

```
Counting objects: 8, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (8/8), 15.78 KiB | 0 bytes/s, done.
Total 8 (delta 0), reused 0 (delta 0)
To https://github.com/fulanodasilva/moveis-ecologicos.git
 * [new branch]      master -> master
```

Se atualizarmos a página do nosso repositório, veremos que os arquivos e os commits foram enviados com sucesso:

Figura 5.17: Página do repositório atualizada com os arquivos e commits

5.7 Clonando o repositório hospedado no GitHub

Agora que o nosso repositório remoto está hospedado no GitHub, os desenvolvedores da equipe poderão sincronizar seus repositórios locais de qualquer lugar do mundo, não ficando mais restritos a sincronizar apenas quando conectados na rede local da empresa.

Para que um novo desenvolvedor obtenha uma cópia do repositório, ele agora deverá clonar o repositório remoto localizado no GitHub:

```
$ git clone https://github.com/fulanodasilva/
moveis-ecologicos.git
```

5.8 Colaborando com projetos open source

Conforme citado anteriormente, existem milhares de projetos hospedados no GitHub, sendo que muitos destes são **open source**, ou seja, significa que qualquer desenvolvedor pode colaborar com melhorias e correções de bugs. Mas como fazer para colaborar com tais projetos?

Vamos supor que queremos colaborar com algum projeto open source, por exemplo, o **VRaptor**, um framework MVC para desenvolvedores Java.

Figura 5.18: Página do repositório do VRaptor

Primeiramente vamos precisar clonar o repositório, para termos acesso ao código-fonte, e então efetuar as melhorias desejadas:

```
$ git clone https://github.com/caelum/vraptor4.git
```

Pronto! Já temos uma cópia do repositório com todos os arquivos do projeto. Agora basta efetuar as melhorias desejadas, registrando as alterações realizadas com commits.

Mas lembre-se de que os commits realizados existem apenas no nosso repositório local, precisamos agora enviá-los para o repositório remoto do projeto, no GitHub, com o comando `git push`:

```
$ git push origin master
```

Entretanto, a execução do comando anterior falha, gerando a seguinte mensagem de erro:

```
remote: Permission to caelum/vraptor4.git denied to
        fulanodasilva.
fatal: unable to access 'https://github.com/caelum/
       vraptor4.git/':
The requested URL returned error: 403
```

O problema é que o usuário `fulanodasilva` não tem permissão para enviar commits para o repositório do VRaptor, pois esse repositório não pertence a ele, mas sim ao usuário `caelum`.

Para resolver esse problema, o usuário `caelum` poderia acessar a página de configurações do repositório VRaptor no GitHub, e adicionar o usuário `fulanodasilva` como **colaborador**.

Mas essa abordagem deve ser utilizada apenas para adicionar os usuários que fazem parte da equipe de desenvolvimento do projeto, ou seja, apenas os desenvolvedores que trabalham efetivamente no projeto. Dessa forma, acaba sendo inviável ter que adicionar um novo usuário como colaborador toda vez que um novo desenvolvedor quiser colaborar com o projeto.

Além disso, a partir do momento em que adicionamos um usuário como colaborador, ele passa a ter permissão de enviar commits para o repositório sem restrições, o que pode ser arriscado, pois não há uma avaliação prévia das alterações que foram realizadas.

Pull requests

A solução do GitHub para a colaboração com projetos open source, foi uma funcionalidade chamada de **pull requests**.

Primeiramente, devemos acessar a página do projeto que queremos colaborar no GitHub, no nosso caso http://github.com/caelum/vraptor4, e clicar no botão **Fork**:

Figura 5.19: Botão para fazer o fork do VRaptor

Ao clicar no botão **Fork**, o GitHub criará uma cópia do repositório do VRaptor no nosso usuário do GitHub:

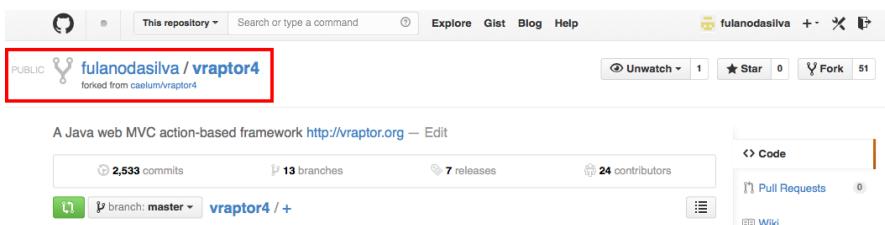



Figura 5.20: Página do repositório do VRaptor pertencente ao nosso usuário

Repare que agora o nosso usuário do GitHub possui um repositório chamado `vraptor4`, ou seja, esse repositório é a **nossa** versão do projeto.

O próximo passo é clonar o repositório a partir do **nosso** usuário do GitHub, e não do usuário `caelum`, pois assim conseguiremos realizar o `push` dos commits locais para o nosso repositório no GitHub:

```
$ git clone https://github.com/fulanodasilva/vraptor4.git
```

Agora, após realizar as alterações e commits no projeto, conseguiremos realizar o `push` normalmente, pois o repositório remoto no GitHub pertence ao nosso usuário.

Entretanto, os commits foram enviados apenas para o **nosso** repositório no GitHub.

Para que os commits sejam integrados ao repositório original do projeto, no caso o repositório `vraptor4` do usuário `caelum`, devemos enviar um **Pull Request** para o usuário `caelum`, solicitando que seja feito um **pull** dos nossos commits.

Para criar um `Pull Request` devemos acessar a página do nosso repositório no GitHub, clicar no link **Pull Requests**, e em seguida clicar no botão **New pull request**:

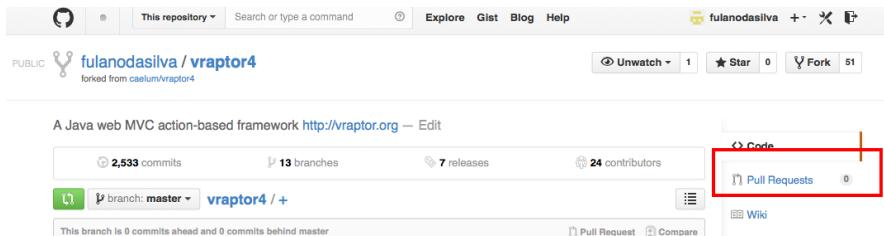



Figura 5.21: Página com link para gerenciar Pull requests

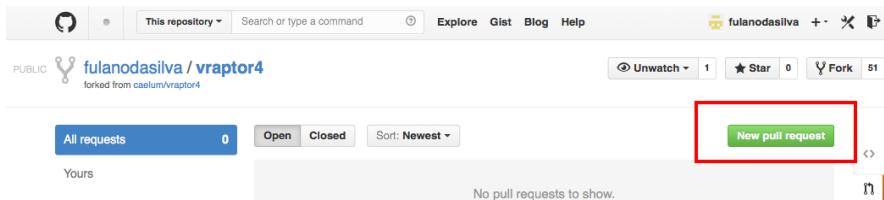



Figura 5.22: Página com botão para criar um novo Pull request

Seremos redirecionados para a página de criação do pull request, na qual serão listados os commits e alterações realizados. Para confirmar a criação do pull request, basta clicar no botão `Create pull request`:

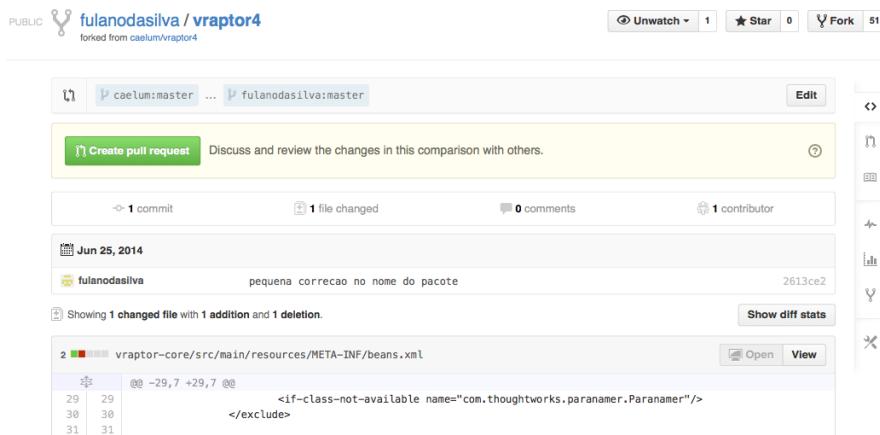



Figura 5.23: Página com botão para confirmar a criação do pull request

Após criarmos o `pull request`, o usuário `caelum` receberá uma notificação do GitHub, e poderá efetuar o `pull` dos nossos commits, integrando-os ao repositório original do projeto. Claro, isso após uma análise das alterações efetuadas em nossos commits.

Pronto! Esse é o processo utilizado pelos desenvolvedores que colaboram com projetos open source hospedados no GitHub.

Capítulo 6

Organizando o trabalho com branches

Nosso cliente não está satisfeito com o *design* da página. Convenhamos, a página que criamos não está das mais bonitas do mundo:

Figura 6.1: Design da página deixa a desejar

Sabemos um pouquinho sobre design, mas precisamos de algumas semanas focadas na melhoria do visual.

Mas temos um problema: nosso cliente continua pedindo para alterarmos o conteúdo da página, várias vezes por semana. Além disso, temos que corrigir *bugs* como erros de português, links incorretos etc. E sabemos que todas essas as mudanças têm que ser publicadas de imediato.

Se publicarmos o site enquanto o novo design está sendo desenvolvido, teremos uma página pela metade. E precisamos de tempo para experimentar algumas possibilidades. E agora?

Trabalhando em paralelo com branches

A maioria dos sistemas de controle de versão permite trabalho em paralelo através de **branches**. Uma *branch* é uma linha independente de desenvolvimento em que podemos comitar novas versões do código sem afetar outras branches.

Em muitos sistemas de controle de versão, utilizar branches é algo bastante lento e trabalhoso. Porém, a estrutura interna do Git permite lidarmos com branches de maneira muito rápida e leve.

6.1 A branch master

Vamos nos certificar de que estamos no nosso repositório `moveis`, executando `cd ~/moveis` em um terminal.

Para listar as branches do nosso repositório, devemos executar o comando:

```
$ git branch
```

Teremos como resposta:

```
* master
```

Peraí! Foi mostrada uma branch chamada `master`, mesmo sem termos criados nenhuma branch... Isso ocorreu porque o Git possui por padrão uma branch principal chamada `master`. Todos os nossos commits até agora ocorreram nessa branch principal.

Observe que a branch `master` tem um asterisco (*) na frente. Isso indica que é a branch atual, em que estamos trabalhando.

No Git, toda branch é basicamente um apontador para um commit. A cada novo commit que fazemos, a branch é movida automaticamente, passando a apontar para esse novo commit. Por isso, por padrão, a branch `master` aponta para o último commit que fizemos.

Se quisermos listar as branches existentes no nosso repositório com os commits associados, poderíamos utilizar a opção `-v` do comando `git branch`:

```
$ git branch -v
```

A resposta, no nosso caso, seria:

```
* master b92285b Revert "Adicionando texto peculiar"
```

Antes de continuarmos, é importante sabermos uma das informações que um commit armazena, o commit pai, que é o que foi efetuado anteriormente. Na verdade, um commit pode ter vários pais, o que veremos adiante.

Podemos observar os commits com seus respectivos pais executando o comando `git log` com a opção `--parents`.

Já para verificarmos o commit para o qual a `master` está apontando, passamos a opção `--decorate` para o comando `git log`.

Juntando as opções, podemos executar `git log --oneline --decorate --parents` para exibir o histórico resumido do nosso repositório com o commit para o qual a `master` está apontado e os commits pai de cada commit. Teríamos uma saída parecida com:

```
b92285b a5b9fce (HEAD, master) Revert "Adicionando texto
                                       peculiar"
a5b9fce 2259f55 Adicionando texto peculiar
2259f55 d211fac Movendo principal.js
d211fac 515da0a Renomeando CSS
515da0a e9ca4c6 Removendo página de produtos
e9ca4c6 793e567 Página de produtos
793e567 0f2b749 Diminuindo intervalo de troca de banner
0f2b749 7e1c195 Banner ao abrir pagina
```

```
7e1c195 9a20490 Inserindo arquivo principal.js
9a20490 83b60c3 Script de troca de banner
83b60c3 1a02a4f Inserindo titulo e diminuindo tamanho da pagina
1a02a4f Commit inicial
```

Observe que a resposta do comando anterior apresenta três colunas: o código do commit, o código do pai e a mensagem do commit. Por exemplo, podemos ver que o pai do commit `d211fac` ("Renomeando CSS") foi o commit `515da0a` ("Removendo página de produtos").

Note também que nosso primeiro commit, o de código `1a02a4f`, possui apenas duas colunas, indicando que esse commit não possui um pai.

Além disso, podemos ver que a branch `master` aponta para o último commit, de código `b92285b`. Não se preocupe com o `HEAD`, já que veremos a seguir seu significado.

Uma visualização do nosso histórico de commits da branch `master` seria:

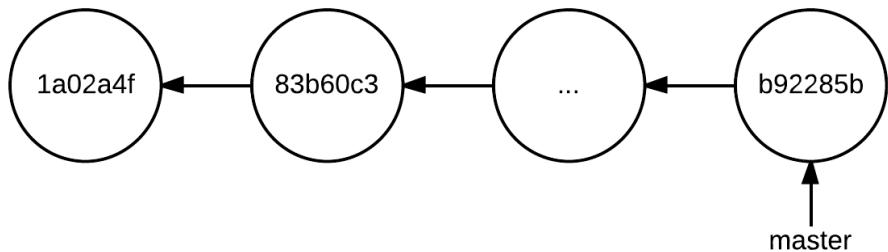



Figura 6.2: A branch master

6.2 Criando uma branch

Para criarmos uma branch chamada `design` para trabalharmos na melhoria do design basta executarmos:

```
$ git branch design
```

Não será exibida nenhuma resposta.

Se listarmos as branches com o comando `git branch`, teremos:

```
design
* master
```

Agora temos duas branches!

Note que a branch atual continua sendo a `master`, que está destacada com um asterisco (*). O comando `git branch` apenas cria uma nova branch, mas não muda para a branch criada.

A nova branch que acabamos de criar (`design`) aponta para o mesmo commit que a branch que estávamos anteriormente (`master`). Podemos verificar isso com o comando `git branch -v`:

```
design b92285b Revert "Adicionando texto peculiar"
* master b92285b Revert "Adicionando texto peculiar"
```

Mas onde o Git guarda a informação de qual é a branch atual? Aí que entra o `HEAD`, que é um apontador especial que indica qual é a branch na qual estamos trabalhando.

Visualizando o nosso repositório logo após a criação da branch `design`, teríamos:

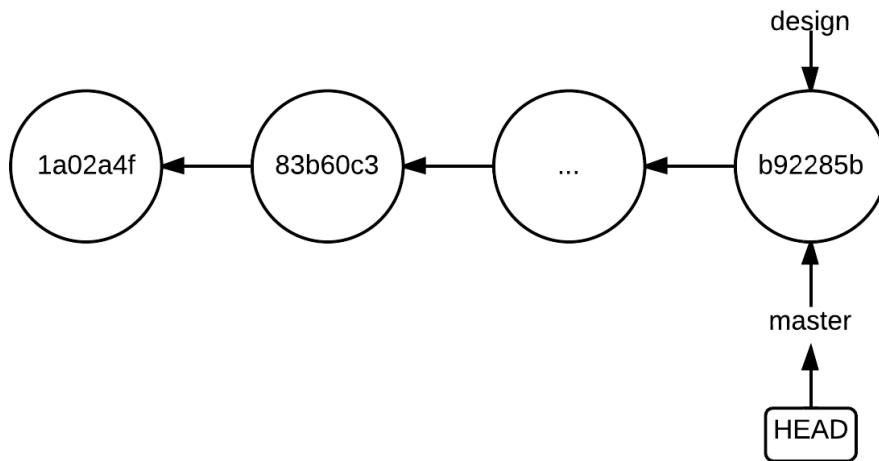



Figura 6.3: A branch design e o HEAD

6.3 Trocando de branch

Já criamos nossa branch `design` mas ainda estamos na `master`. Para trocarmos para a branch recentemente criada, devemos executar:

```
$ git checkout design
```

Deve aparecer como resposta:

```
Switched to branch 'design'
```

Depois de executar o comando anterior, o `HEAD` passa a apontar para a nova branch `design`.

Se executarmos o comando `git branch -v`, teremos:

```
* design b92285b Revert "Adicionando texto peculiar"
  master b92285b Revert "Adicionando texto peculiar"
```

Observe que o asterisco (*) que indica a branch atual foi mudada para a branch `design`. Ambas as branches continuam apontando para o mesmo commit.

Visualizando o estado do nosso repositório, teríamos:

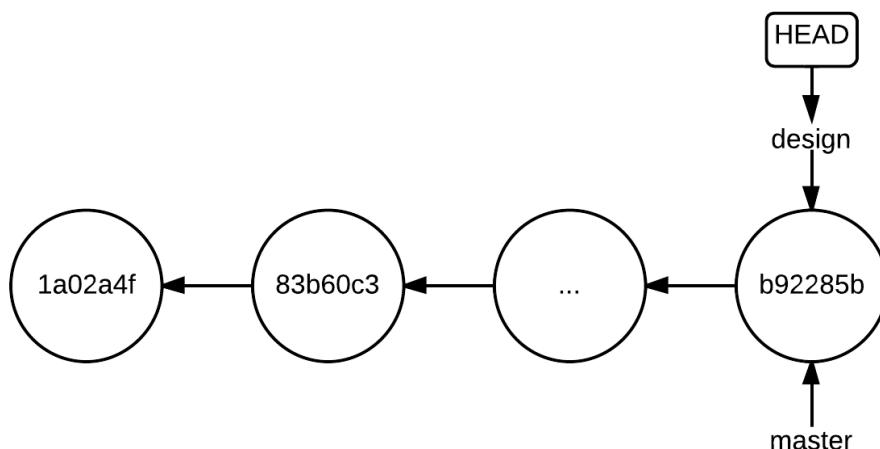



Figura 6.4: HEAD apontando para design

Conforme vimos no capítulo 3, o comando `git checkout`, além de trocar a branch atual, também pode ser usado para descartar mudanças de um arquivo que ainda não estão na área de stage. Para descartarmos as mudanças do arquivo `index.html`, por exemplo, devemos executar `git checkout -- index.html`.

Na verdade, a opção `--` não é estritamente necessária. Porém, é uma boa prática, já que serve para informar que se trata de um arquivo. Dessa maneira, ajuda a evitar problemas caso uma branch tenha o mesmo nome de algum arquivo.

Criando e já trocando para uma nova branch

Nosso cliente pediu que nós começássemos a construir uma loja online para venda dos móveis. Seria algo demorado, de longo prazo. Por isso, construiríamos a loja online uma nova branch, chamada `loja`.

Criar uma branch `loja` com `git branch` só para depois mudar para a nova branch com `git checkout` é algo tedioso.

Seria interessante já criarmos e mudarmos para a nova branch `loja` de uma só vez. Podemos fazer isso passando a opção `-b` para o comando `git checkout`:

```
$ git checkout -b loja
```

A saída apresentada seria:

```
Switched to a new branch 'loja'
```

Se listarmos as branches com `git branch -v`, teríamos:

```
design b92285b Revert "Adicionando texto peculiar"
* loja   b92285b Revert "Adicionando texto peculiar"
  master b92285b Revert "Adicionando texto peculiar"
```

Note que o asterisco (*) está na branch `loja`, indicando que essa é a branch atual. E a `loja` foi criada apontando para o commit `b92285b`, o mesmo das outras branches.

6.4 Deletando uma branch

Mas não temos tempo para criar uma loja online... Temos que nos focar nas nossas tarefas atuais.

Como não vamos usar a branch `loja` por um bom tempo, seria bom removê-la. Assim, evitamos confusões e desperdício no nosso repositório.

Não é possível remover uma branch enquanto estivermos nela. Por isso, devemos ir para outra branch. Para ir para a branch `master`, devemos executar `git checkout master`.

Para deletar uma branch, devemos utilizar a opção `-d` do comando `git branch`:

```
$ git branch -d loja
```

Deve aparecer uma mensagem como a seguir:

```
Deleted branch loja (was b92285b).
```

A branch `loja` foi deletada. Nosso repositório está limpo e cristalino!

Se já tivéssemos feito algum commit na branch `loja`, ao executarmos `git branch -d loja` teríamos como resposta:

```
error: The branch 'loja' is not fully merged.
If you are sure you want to delete it, run 'git branch -D loja'.
```

Não é possível deletar com a opção `-d` uma branch que possui commits ainda não aplicados em outras branches (veremos como mesclar branches mais adiante).

Para removermos a branch `loja` se tivermos feito algum commit, devemos utilizar a opção `-D`:

```
$ git branch -D loja
```

Teríamos na saída:

```
Deleted branch loja (was b92285b).
```

Agora sim! Os commits da branch `loja` seriam descartados e a branch seria removida.

6.5 Comitando código em uma nova branch

Decidimos utilizar uma ferramenta para auxiliar na melhoria do design: o **Bootstrap**.

A primeira coisa é certificar-nos que estamos na branch `design`, executando `git checkout design`.

Depois disso, vamos alterar o `index.html`, trocando nosso CSS feio pelo CSS do Bootstrap e colocando o atributo `class` em algumas tags:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <meta name="description" content="Móveis ecológicos">
        <meta name="keywords" content="moveis ecologicos">
        <link rel="stylesheet"
           href="http://netdna.bootstrapcdn.com/bootstrap/3.1.1/
                                                css/bootstrap.min.css">
        <title>Móveis Ecológicos</title>
        <script src="js/principal.js"></script>
    </head>
    <body onload="trocaBanner();">
        <div class="container">
            <div class="jumbotron">
                <h1>Móveis Ecológicos S. A.</h1>
                <h2 id="mensagem"></h2>
                <ul class="list-group">
                    <li class="list-group-item">
                        Móveis de garrafas PET
                    </li>
                    <li class="list-group-item">
                        Móveis de latinhas de alumínio
                    </li>
                    <li class="list-group-item">
                        Móveis de papelão
                    </li>
                </ul>
            </div>
        </div>
```

```
    </body>
</html>
```

Apenas com essas pequenas alterações, embelezamos um bocado nossa página:

Figura 6.5: Página embelezada

Também vamos remover o nosso CSS antigo, para não ficarmos com código obsoleto, executando o comando `git rm principal.css`.

Agora, vamos adicionar à área de stage as alterações e comitá-las com o comando `git commit -am "Adicionando Bootstrap"`.

Na saída, será impresso:

```
[design 8f53065] Adicionando Bootstrap
 2 files changed, 23 insertions(+), 19 deletions(-)
 delete mode 100644 principal.css
```

Observe no texto anterior que foi exibido o nome da branch `design`, em que estamos comitando.

Ao executarmos o comando `git log -n 3 --oneline --decorate --parents`, teríamos:

```
8f53065 b92285b (HEAD, design) Adicionando Bootstrap
b92285b a5b9fce (master) Revert "Adicionando texto peculiar"
a5b9fce 2259f55 Adicionando texto peculiar
```

Note que a branch `master` ainda continua apontando para o commit de código `b92285b` (Revert "Adicionando texto peculiar").

Já a branch `design` passou a apontar para o novo commit, de código `8f53065` (Adicionando Bootstrap). Observe também que o `HEAD` aponta para o mesmo commit.

Um gráfico representando o nosso commit na branch `design` seria:

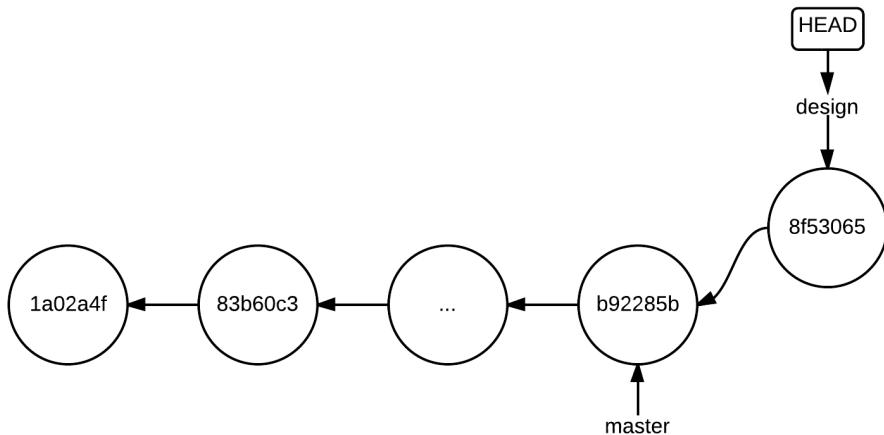



Figura 6.6: Commit na branch design

6.6 Voltando para o master e fazendo uma alteração

Ainda não acabamos de embelezar a nossa página, mas nosso cliente nos ligou pedindo que modificássemos os textos do banner. E pediu que publicássemos urgentemente as alterações!

A primeira coisa que devemos fazer é voltar para a branch `master`, executando o comando `git checkout master`.

Visualizando o estado do nosso repositório nesse ponto, teríamos:

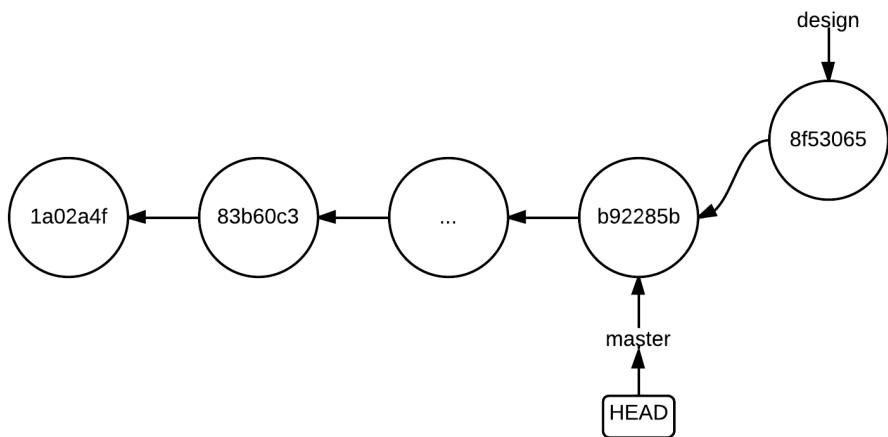



Figura 6.7: Voltando à branch master

Agora que estamos na branch `master`, vamos alterar os textos do banner no arquivo `principal.js`:

```
var banners = ["Do lixo ao luxo!","Reaproveitar é aproveitar!"];
//restante do arquivo
```

Em seguida, vamos comitar nossas alterações com o comando `git commit -am "Alterando textos do banner"`. Teríamos na saída:

```
[master 2223859] Alterando textos do banner
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Nesse ponto, nosso repositório estaria parecido com:

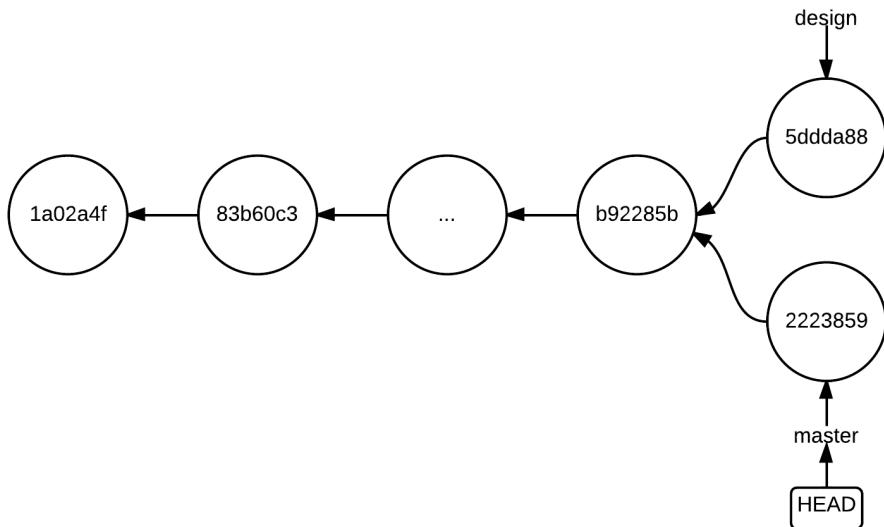



Figura 6.8: Novo commit na master

Note que a branch `master` está apontando para o commit `2223859` que acabamos de realizar. Já a branch `design` continua apontando para o mesmo commit que estava anteriormente.

Poderíamos então publicar com rapidez os novos banners sem as modificações de design, que ainda não foram aprovadas.

6.7 Mesclando alterações

Mostramos o novo design para o nosso cliente e ele gostou bastante. Pediu para colocarmos no ar o que temos agora. E o mais rápido possível!

Temos duas alterações: os novos textos do banner, que estão na branch `master`, e o embelezamento da página, que está na branch `design`.

Por isso, precisamos mesclar as alterações que fizemos na branch `design` com as que fizemos na `master`. Identificar o que mudou, copiando e colando na mão é um trabalho razoável, com alta chance de inserirmos um erro. Ainda bem que o Git vai nos ajudar a fazer isso!

Verificando branches ainda não mescladas

Considerando que estamos na branch `master`, podemos verificar as branches ainda não mescladas com a opção `--no-merged` do comando `git branch`:

```
$ git branch --no-merged
```

Teríamos a seguinte saída:

```
design
```

Isso indica que há mudanças ainda não mescladas na branch `design`.

O comando `git branch` também tem a opção `--merged`. Se executássemos `git branch --merged`, teríamos:

```
* master
```

Mesclando alterações com merge

Para juntarmos todas as alterações que fizemos na branch `design` com as da branch `master`, mesclando as duas, podemos utilizar o comando:

```
$ git merge design -m "Mesclando com a branch design"
```

Será exibida uma resposta semelhante à seguinte:

```
Removing principal.css
Merge made by the 'recursive' strategy.
 index.html    | 31 +++++++++++++++++++++++--------
 principal.css | 11 -----------
 2 files changed, 23 insertions(+), 19 deletions(-)
 delete mode 100644 principal.css
```

Pronto! Mudanças mescladas! Ao executarmos o comando `git branch --no-merged`, não teríamos nenhuma resposta, indicando que não há nenhuma branch não mesclada.

Note que, depois do merge, o arquivo `principal.css` foi removido e as alterações que tínhamos realizado em `index.html` foram trazidas para

a nossa branch atual, a `master`. Observe também que os textos do banner que modificamos continuam lá.

Se não tivéssemos informado uma mensagem através da opção `-m`, seria aberto um editor de texto.

É importante saber que, quando é efetuado um merge, é criado um novo commit com as alterações que estamos mesclando e com a mensagem que informamos.

Podemos verificar esse novo commit de merge executando o comando `git log -n 4 --oneline --decorate --parents`. Teríamos na saída:

```
e6b2f35 2223859 8f53065 (HEAD, master) Mesclando com a branch
                                                          design
2223859 b92285b Alterando textos do banner
8f53065 b92285b (design) Adicionando Bootstrap
b92285b a5b9fce Revert "Adicionando texto peculiar"
```

Observe que a branch `master` e o `HEAD` apontam para o commit de merge, que é o último commit realizado, e que tem o código `e6b2f35` e a mensagem que havíamos informado.

Já a branch `design` continua apontando para o mesmo commit que estava anteriormente.

É importante notar que o commit de merge (`e6b2f35`) tem dois commits pai: o commit `2223859`, de mensagem "Alterando textos do banner", e o commit `8f53065`, de mensagem "Adicionando Bootstrap".

Visualizando o estado do nosso repositório depois do merge da branch `design` na branch `master`, teríamos:

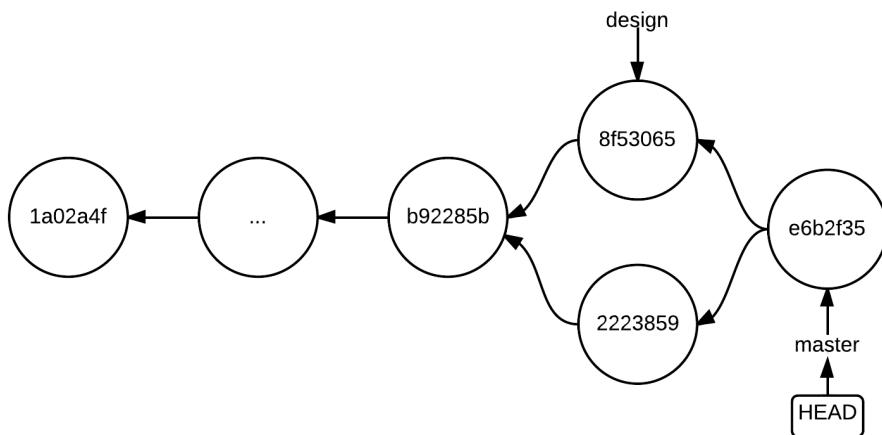



Figura 6.9: Merge de design na master

Há uma situação em que um `git merge` não irá gerar um commit de merge: quando a branch de destino não teve nenhum commit a mais desde que a branch a ser mesclada foi criada.

No nosso caso, se não tivéssemos feito o commit `2223859` (Alterando textos do banner) na branch `master`, o merge com a branch `design` seria basicamente apontar a `master` para onde a `design` aponta: o commit `8f53065` (Adicionando Bootstrap).

Esse tipo de merge é chamado de *fast-forward*.

É possível evitar o uso de fast-forward, forçando a criação de um commit de merge, utilizando a opção `--no-ff` do comando `git merge`.

Mesclando alterações com rebase

À medida que uma aplicação vai sendo desenvolvida, é natural isolarmos partes do trabalho em branches. E quando algo estiver pronto, será feito um merge do código e, por isso, teremos um novo commit de merge.

Para projetos com muitas branches e muitos merges, o histórico do repositório ficará repleto de commits de merge. Isso pode tornar bastante traba-

lhosas tarefas como revisão de código.

Há uma maneira *alternativa* de mesclar as alterações de duas branches que simplifica o histórico do projeto. Considerando que estamos na branch `master`, em vez de utilizar `git merge`, é possível utilizar:

```
$ git rebase design
```

Na saída, deve aparecer algo como:

```
First, rewinding head to replay your work on top of it...
Applying: Alterando textos do banner
```

Pronto! Outra maneira de termos as mudanças mescladas! A resposta do comando `git branch --no-merged` seria vazia, já que não há nenhuma branch não mesclada.

As mensagens anteriores informam que o repositório foi rebobinado e nosso trabalho refeito.

No nosso caso, o repositório voltou ao estado que estava ao criarmos a branch `design` a partir da `master`.

Depois disso, foi aplicado o commit que fizemos na branch `design`.

Finalmente, foi aplicado o nosso último commit na `master`, as alterações no texto dos banners.

Ao verificarmos o histórico do nosso repositório com o comando `git log -n 3 --oneline --decorate --parents`, teríamos:

```
c05c05a 8f53065 (HEAD, master) Alterando textos do banner
8f53065 b92285b (design) Adicionando Bootstrap
b92285b a5b9fce Revert "Adicionando texto peculiar"
```

A branch `design` continua apontando para o commit que estava anteriormente.

Já a branch `master` teve seu histórico refeito: foi criado um **novo commit** com as alterações no texto dos banners, com código `c05c05a`.

Observe que o pai do novo commit `c05c05a` (Alterando textos do banner) é o commit `8f53065` (Adicionando Bootstrap), que tínhamos feito na branch design.

Perceba que não houve commit de merge. O histórico de commits foi linearizado.

Colocando o estado do nosso repositório após o rebase em um gráfico, teríamos:

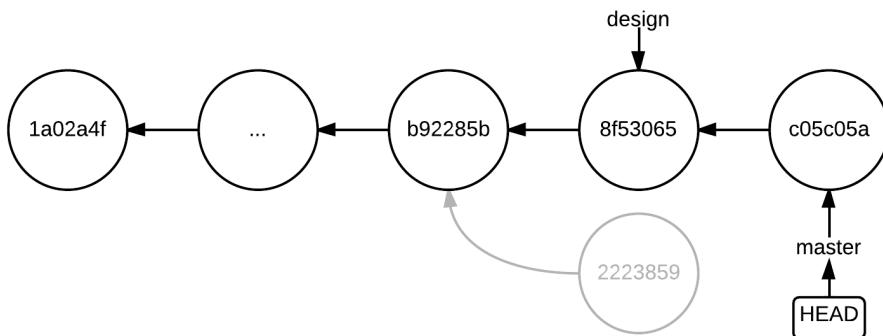



Figura 6.10: Rebase de design na master

No caso de um fast-forward basta apontar a branch de destino para o commit mais novo da branch sendo mesclada.

No nosso caso, se não tivéssemos nenhum novo commit na branch `master`, a mesclagem seria feita apontando a `master` para o mesmo commit para o qual a `design` aponta. Dessa forma, o rebase não precisaria reescrever o histórico de commits.

Por isso, o `git merge` e o `git rebase` têm o mesmo efeito quando há um fast-forward.

Mantendo o histórico da master intacto após um rebase

Evitar ao máximo alterações no histórico da `master` é uma prática bastante adotada por usuários do Git.

Porém, nosso rebase anterior acabou deixando obsoleto o commit `2223859` (Alterando textos do banner), substituindo-o pelo commit `c05c05a`, que tem o mesmo conteúdo.

Poderíamos ter feito a integração das branches `master` e `design` com rebase de maneira a deixar `master` intacta se tivéssemos feito nosso trabalho de uma maneira ligeiramente diferente.

Primeiramente, teríamos que passar para a branch `design` com o comando `git checkout design`.

Então, faríamos o rebase da branch `master` na `design` com o comando `git rebase master`.

Após o rebase, o commit `8f53065` (Adicionando Bootstrap) da branch `design` foi substituído, digamos, pelo commit de código `e887c10`.

Visualizando graficamente nosso repositório nesse momento, teríamos:

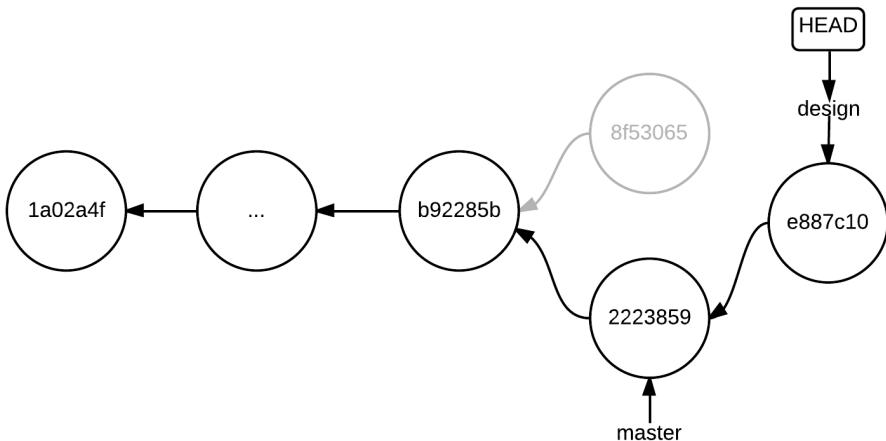



Figura 6.11: Rebase de master na design

É como se refizéssemos o commit base da branch `design`, o commit da branch `master` que é ancestral do commit que fizemos em `design` (e de todos que virão).

Antes do rebase, o commit base era o de código `b92285b` (Revert "Adicionando texto peculiar").

Depois que o rebase foi executado, o commit base passou a ser o último commit da `master`, de código `2223859` (Alterando textos do banner).

O nome *rebase* vem da ideia de refazer o commit base de uma branch.

Para isso, o Git precisa criar novos commits, mudando os ancestrais de commits anteriores.

Com a base da branch `design` refeita, é possível fazer um merge do tipo fast-forward da `design` na branch `master`. Para tal, deveríamos ir para a `master` executando `git checkout master`. Então, faríamos um `git merge design`. O gráfico do repositório ficaria parecido com:

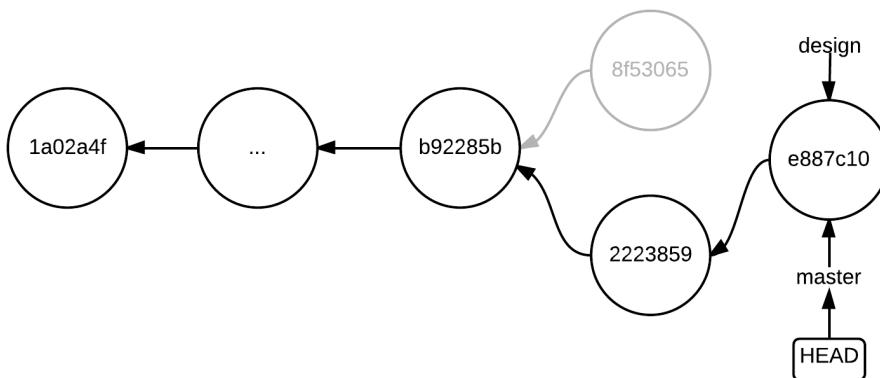



Figura 6.12: Merge fast-forward de design na master, após o rebase

Para saber mais: Qual usar? Um merge ou um rebase?

Utilizar merge mantém um registro fiel do que ocorreu com o nosso repositório, mas os commits de merge complicam tarefas como navegar pelo código antigo e revisar código novo.

Já o rebase simplifica o histórico, mas perdemos informação sobre nosso repositório e alguns commits são reescritos. No caso de conflitos, as coisas podem ficar especialmente complicadas.

Não há uma resposta simples sobre a maneira ideal de mesclarmos alterações com Git. Cada solução tem seus prós e contras.

Ao trabalharmos com branches remotas, nosso próximo assunto, surgem novas questões.

Capítulo 7

Trabalhando em equipe com branches remotas

Na dia a dia, dificilmente trabalhamos sozinhos. No capítulo 4 vimos como trabalhar com repositórios remotos e no capítulo 5 vimos como utilizar o GitHub, o que nos permite trabalhar em equipe de maneira consistente.

Trabalhar com branches, que vimos no capítulo 6, também é uma boa maneira de organizar o trabalho em equipe. Mas quando trabalhamos com repositórios remotos, surgem novas preocupações.

Lembrando dos repositórios remotos

No nosso repositório local `moveis`, temos o remote `origin` que aponta para o GitHub, o que podemos verificar com o comando `git remote -v`:

```
origin  git@github.com:fulanodasilva/moveis-ecologicos.git
```

```
(fetch)
origin  git@github.com:fulanodasilva/moveis-ecologicos.git
(push)
```

O remote `origin` está apontando para um repositório lá do GitHub, que também tem sua branch `master`. Precisamos de uma maneira de receber e enviar commits para a branch `master` do GitHub.

7.1 Branches remotas

No Git, temos **branches remotas** que apontam para branches que estão nos repositórios remotos configurados.

Para diferenciá-las das branches locais, o nome de uma branch remota é o nome do remote seguido do nome da branch. No nosso caso, a branch `master` lá do GitHub tem o nome de `origin/master`, já que o remote `origin` aponta para o GitHub.

Ao contrário de branches locais, que são movidas a cada commit, branches remotas só são movidas ao fazermos operações que envolvam comunicação de rede como `git push`, `git pull` e também o comando `git fetch`, que veremos mais adiante. Portanto, uma branch remota representa a situação da última vez que houve comunicação com o repositório remoto.

Podemos listar as branches remotas passando a opção `-r` para o comando `git branch`:

```
$ git branch -r
```

Teremos na saída:

```
origin/master
```

> Se quisermos mostrar tanto as branches locais como as remotas, podemos utilizar o comando `git branch -a`.

Para vermos para quais commits as branches remotas estão apontando, podemos utilizar o comando anterior com a opção `-v`:

```
$ git branch -r -v
```

Teremos:

```
origin/master b92285b Revert "Adicionando texto peculiar"
```

Observe que a branch `origin/master` está apontando o commit de código `b92285b` (Revert "Adicionando texto peculiar").

Considerando que fizemos o merge da branch `master` com a branch `design` no repositório local no capítulo 6, ao executarmos o comando `git log -n 4 --oneline --decorate --parents`, teríamos:

```
e6b2f35 2223859 8f53065 (HEAD, master) Mesclando com a branch
                                                        design
2223859 b92285b Alterando textos do banner
8f53065 b92285b (design) Adicionando Bootstrap
b92285b a5b9fce (origin/master) Revert "Adicionando texto
                                        peculiar"
```

Observe que a branch `origin/master` aponta para um commit anterior às branches `master` e `origin` do repositório local.

Exibindo gráficos com o repositório lá do GitHub e o repositório local, cada um com suas branches, ficaríamos com algo parecido com:

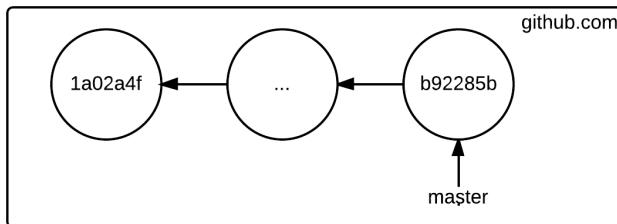



Figura 7.1: O repositório do GitHub não contém a branch design, apenas a sua master

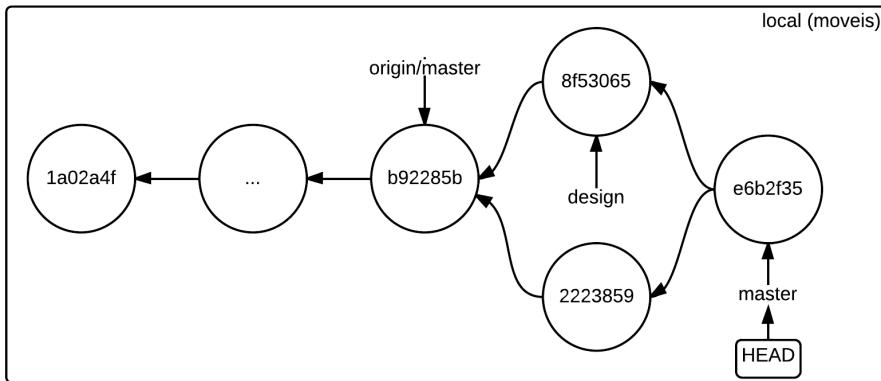



Figura 7.2: Repositório local tem as branches locais master e design e a remota origin/master

No fim das contas, tanto a branch `master` lá do GitHub como a nossa branch `origin/master` apontam para o commit de código `b92285b` (Revert "Adicionando texto peculiar").

7.2 Compartilhando branches

A branch `design` ainda não existe lá no GitHub. Para compartilhá-la devemos informar os nomes do remote e da branch para o comando `git push`:

```
$ git push origin design
```

Deverá aparecer uma resposta parecida com:

```
Counting objects: 5, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 669 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@github.com:fulanodasilva/moveis-ecologicos.git
 * [new branch]      design -> design
```

O comando anterior envia para o remote `origin` os commits da branch `design`.

Depois disso, podemos ver a branch `design` na página do nosso projeto no GitHub.

> Também podemos ver um gráfico das branches do repositório na URL https://github.com/fulanodasilva/moveis-ecologicos/network (não esqueça de trocar `fulanodasilva` pelo seu usuário.)

Nesse momento, se listarmos as branches remotas e locais com o comando `git branch -a -v`, teríamos:

```
design                  8f53065 Adicionando Bootstrap
* master                  e6b2f35 Mesclando com a branch design
  remotes/origin/design 8f53065 Adicionando Bootstrap
  remotes/origin/master b92285b Revert "Adicionando texto
                                              peculiar"
```

Observe que temos as branches locais `master` e `design` e as branches remotas `origin/master` e `origin/design`.

Como acabamos de fazer o `git push` da branch `origin/design`, essa branch remota e a branch local `design` estão apontando para o commit de código `8f53065` (Adicionando Bootstrap).

Já a branch local `master` e a remota `origin/master` estão apontando para commits diferentes.

Colocando o estado atual dos nossos repositórios local e do GitHub em gráficos, teríamos:

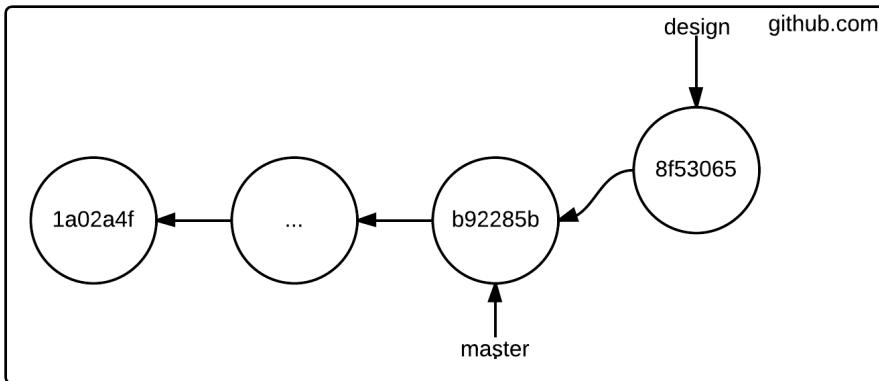



Figura 7.3: Branch design no GitHub

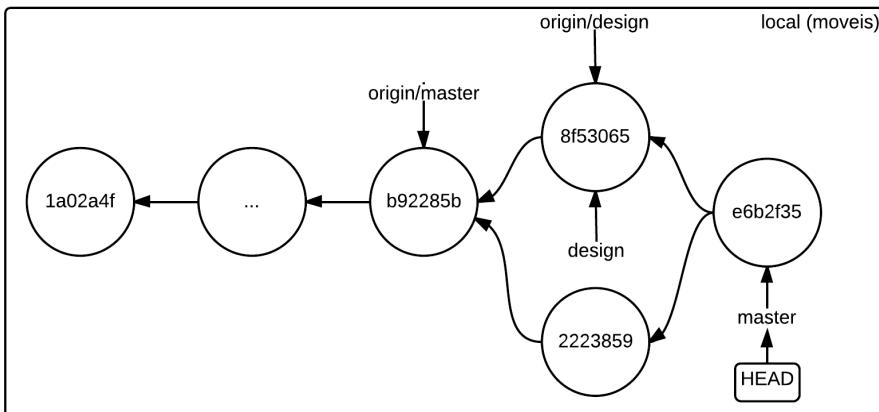



Figura 7.4: Branch remota origin/design no repositório moveis

7.3 Obtendo novas branches remotas em outros repositórios

Para simular um outro membro do nosso time, vamos baixar o repositório em uma outra pasta.

Primeiramente, devemos voltar à nossa pasta pessoal, executando `cd ~`.

Então, vamos clonar o repositório em um diretório chamado `outro`, executando o comando `git clone git@github.com:fulanodasilva/moveis-ecologicos.git outro`.

Depois de clonar o repositório, é importante entrar no diretório, com o comando `cd outro`.

Se listarmos as branches locais e remotas com o comando `git branch -a -v`, teremos:

```
* master                b92285b Revert "Adicionando texto
                                        peculiar"
  remotes/origin/design 8f53065 Adicionando Bootstrap
  remotes/origin/master b92285b Revert "Adicionando texto
                                          peculiar"
```

Note que temos a branch local `master` e as remotas `origin/master` e `origin/design`.

Porém, não temos ainda uma branch local `design`. Se precisarmos trabalhar nessa branch no novo repositório `outro`, precisamos executar o seguinte comando:

```
$ git checkout -b design origin/design
```

O comando apresentará na saída algo como:

```
Branch design set up to track remote branch design from origin
by rebasing.
Switched to a new branch 'design'
```

Com o comando anterior, fizemos um `git checkout` de uma nova branch local chamada `design` a partir da branch remota `origin/design`.

Agora estamos prontos para trabalhar no design da aplicação, se quisermos.

Branches locais criadas a partir de branches remotas são chamadas de *tracking branches*.

Se executarmos `git branch -a -v` novamente, teremos:

```
* design                8f53065 Adicionando Bootstrap
  master                b92285b Revert "Adicionando texto
                                      peculiar"
  remotes/origin/design 8f53065 Adicionando Bootstrap
  remotes/origin/master b92285b Revert "Adicionando texto
                                      peculiar"
```

> Uma outra maneira, mais sucinta, de criar uma tracking branch é utilizar a opção `-t` do comando `git checkout`.
>
> No nosso caso, faríamos `git checkout -t origin/design`. Seria criada localmente a tracking branch `design` relacionada com a branch remota `origin/design`.

Depois de criada a tracking branch `design` no repositório `outro`, os repositórios moveis e o do GitHub estariam inalterados.

Se visualizarmos o repositório `outro`, teríamos:

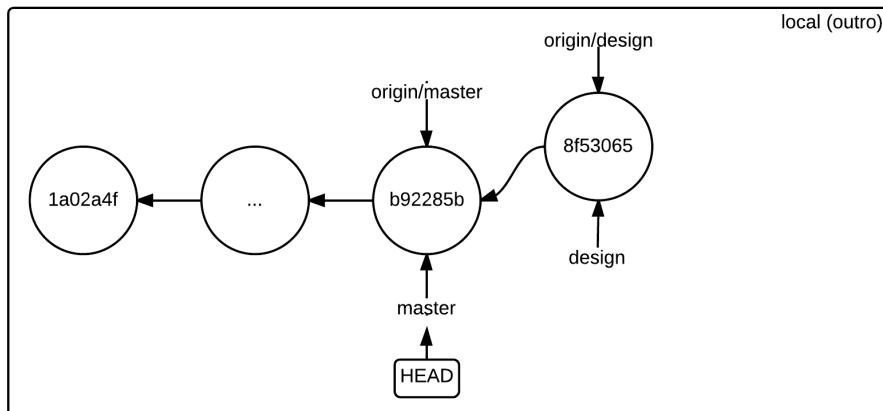



Figura 7.5: Tracking branch design no repositório outro

7.4 Enviando commits para o repositório central

Vamos voltar para nosso diretório `moveis`, executando o comando `cd ~/moveis`.

Nosso repositório `moveis` está dois commits na frente da branch remota `origin/master`.

Podemos verificar isso executando `git status`:

```
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working directory clean
```

Se executarmos, `git log -n 4 --oneline --decorate`, teremos:

```
e6b2f35 (HEAD, master) Mesclando com a branch design
2223859 Alterando textos do banner
8f53065 (origin/design, design) Adicionando Bootstrap
b92285b (origin/master) Revert "Adicionando texto peculiar"
```

Então, para enviar as alterações na branch `master` para o repositório remoto, devemos fazer:

```
$ git push origin master
```

Teremos como resposta algo como:

```
Counting objects: 10, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 670 bytes | 0 bytes/s, done.
Total 6 (delta 3), reused 0 (delta 0)
To git@github.com:fulanodasilva/moveis-ecologicos.git
   b92285b..e6b2f35  master -> master
```

Depois disso, a branch `master` e a `origin/master` estarão apontando para o commit de merge `e6b2f35` (Mesclando com a branch design). É possível nos certificarmos disso, executando `git log -n 4 --oneline --decorate` novamente. Teríamos:

```
e6b2f35 (HEAD, origin/master, master) Mesclando com a branch
                                                        design
2223859 Alterando textos do banner
8f53065 (origin/design, design) Adicionando Bootstrap
b92285b Revert "Adicionando texto peculiar"
```

Já no repositório `outro`, a branch `master` ainda continua apontando para o commit `b92285b` (Revert "Adicionando texto peculiar") e a branch `design` continua apontando para o commit `8f53065` (Adicionando Bootstrap).

O repositório `outro` não sofreu alterações. Já uma visualização do estados do repositórios `moveis` e do GitHub seria:

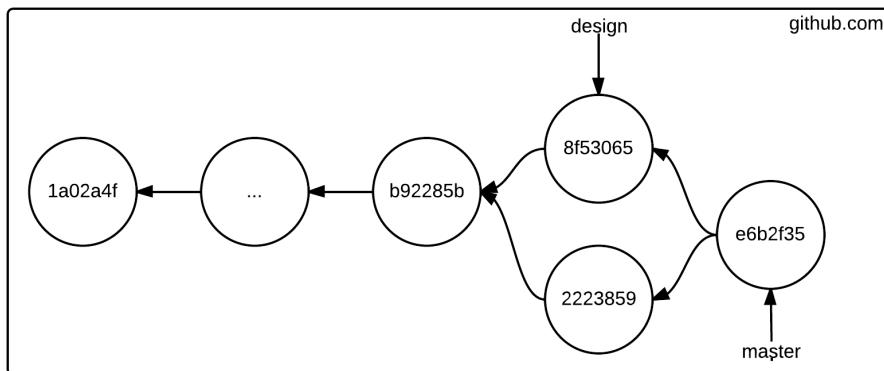



Figura 7.6: Commits de moveis foram recebidos pelo repositório do GitHub

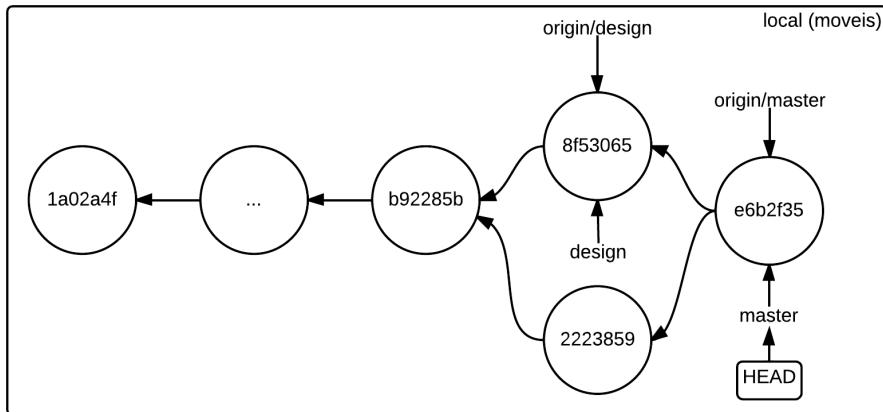



Figura 7.7: Branch origin/master foi atualizada no repositório moveis

7.5 Obtendo commits de uma branch remota

Um novo commit

Digamos que outro desenvolvedor da nossa equipe esteja trabalhando em uma página que lista móveis feitos a partir de garrafas PET. Todo trabalho está sendo feito diretamente na branch `master`.

Para simular esse outro desenvolvedor, vamos utilizar nosso repositório `outro`. Retornaremos a esse repositório, executando o comando `cd ~/outro`.

Então, vamos nos certificar que estamos na branch `master`, executando `git checkout master`.

Feito isso, vamos criar um arquivo `moveis_pet.html` com o seguinte conteúdo:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <meta name="description"
              content="Móveis ecológicos PET">
```

```
        <meta name="keywords" content="moveis ecologicos pet">
        <link rel="stylesheet" href="principal.css">
        <title>Móveis Ecológicos de garrafas PET</title>
    </head>
    <body>
        <h1>Móveis Ecológicos S. A.</h1>
        <h2>Móveis de Garrafas PET</h2>
        <p>Funcionais, baratos e amigos do meio ambiente.</p>
        <ul>
            <li>Sofás</li>
            <li>Racks</li>
            <li>Cadeiras</li>
            <li>Mesas</li>
        </ul>
    </body>
</html>
```

Com o arquivo `moveis_pet.html` criado, vamos rastreá-lo com `git add moveis_pet.html` e gravá-lo no repositório com `git commit -m "Adicionando página sobre móveis PET"`.

Se mostramos o histórico do repositório nesse ponto com `git log -n 2 --oneline --decorate`, teríamos:

```
e1b10ff (HEAD, master) Adicionando página sobre móveis PET
b92285b (origin/master) Revert "Adicionando texto peculiar"
```

Repare que a branch `origin/master` do repositório `outro` está no commit `b92285b` (Revert "Adicionando texto peculiar"), apesar da branch `master` lá do GitHub estar alguns commits na frente, no commit `e6b2f35` (Mesclando com a branch design).

O repositório `moveis` e o do GitHub continuaram iguais. Já se visualizarmos o estado do repositório `outro` após o commit, teríamos:

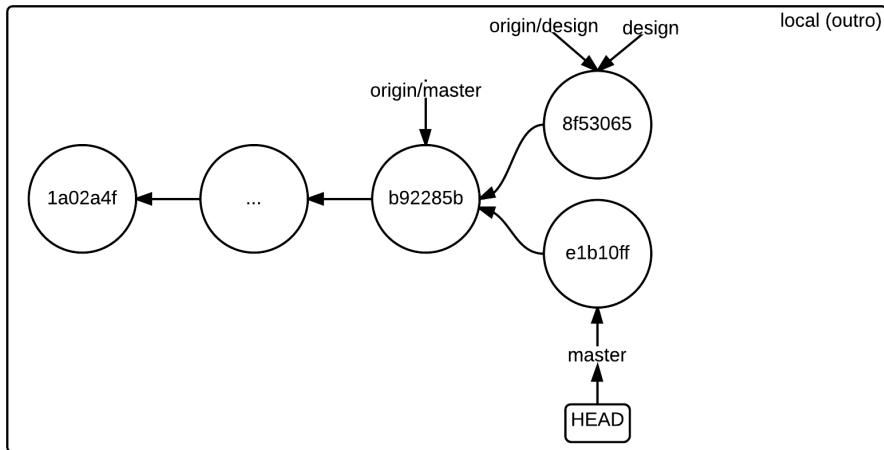



Figura 7.8: Branch master do repositório outro apontando para o novo commit

Trazendo commits de um repositório remoto com fetch

Para obtermos os commits da branch remota `origin/master` no nosso repositório `outro`, podemos executar o comando:

```
$ git fetch origin
```

Teremos como resposta algo como:

```
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From github.com:fulanodasilva/moveis-ecologicos
   b92285b..e6b2f35  master     -> origin/master
```

Com o comando `git fetch`, trazemos os commits de uma branch remota que ainda não estavam presentes localmente.

No nosso caso, obtemos os commits `2223859` (Alterando textos do banner) e `e6b2f35` (Mesclando com a branch design) que estavam lá no GitHub mas ainda não estavam na branch `origin/master`.

Porém, os novos commits da branch remota `origin/master` ainda não foram aplicados na branch local `master`, o que podemos constatar executando o comando `git branch -a -v`:

```
design                  8f53065 Adicionando Bootstrap
* master                  e1b10ff Adicionando página sobre móveis
                                  PET
  remotes/origin/design 8f53065 Adicionando Bootstrap
  remotes/origin/master e6b2f35 Mesclando com a branch design
```

Observe que a branch `master` ainda continua apontando para o commit `e1b10ff` (Adicionando página sobre móveis PET). É importante notar que os novos commits *ainda não foram aplicados* após executarmos o `git fetch`.

Depois de obter os commits da branch `origin/master`, o gráfico do repositório `outro` ficaria:

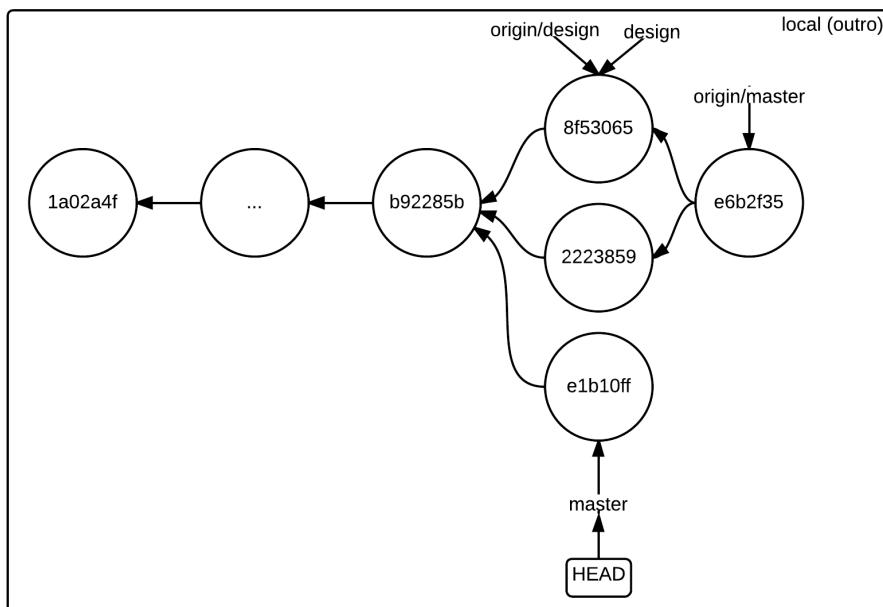



Figura 7.9: Commits remotos obtidos na branch origin/master de outro

7.6 Mesclando branches remotas e locais

Mesclando branches remotas e locais com merge

No repositório `outro`, temos alguns commits na branch remota `origin/master` que ainda não foram aplicados na branch `master`.

Como fazer para mesclar as duas branches? Será que mesclar uma branch remota a uma local precisa de algum comando especial?

Não! Basta usarmos uma das duas maneiras de mesclar branches no Git: um merge ou um rebase.

Se escolhermos fazer um merge e estivermos na branch `master`, basta executarmos:

```
$ git merge origin/master -m "Mesclando origin/master em master"
```

> Se não informarmos uma mensagem para o commit de merge, será aberto um editor de texto.

Na saída, teríamos:

```
Removing principal.css
Merge made by the 'recursive' strategy.
 index.html       | 29 +++++++++++++++++++++--------
 js/principal.js |  2 +-
 principal.css   | 11 -----------
 3 files changed, 22 insertions(+), 20 deletions(-)
 delete mode 100644 principal.css
```

Pronto! Já temos as mudanças lá do GitHub aplicadas na branch local `master` do repositório `outro`.

Se executarmos `git log -n 6 --oneline --decorate --parents`, teremos:

```
550e27d e1b10ff e6b2f35 (HEAD, master) Mesclando origin/master
                                        em master
e1b10ff b92285b Adicionando página sobre móveis PET
```

```
e6b2f35 2223859 8f53065 (origin/master) Mesclando com a branch
                                        design
2223859 b92285b Alterando textos do banner
8f53065 b92285b (origin/design, design) Adicionando Bootstrap
b92285b a5b9fce Revert "Adicionando texto peculiar"
```

> Se não houvesse nenhum novo commit na branch local `master` do repositório `outro`, a `master` teria um *fast-forward*, sendo simplesmente movida para apontar para o mesmo commit da branch remota `origin/master`.

Nenhuma alteração teria acontecido no repositório `moveis` e do GitHub. Agora, um gráfico do histórico anterior do repositório `outro` seria:

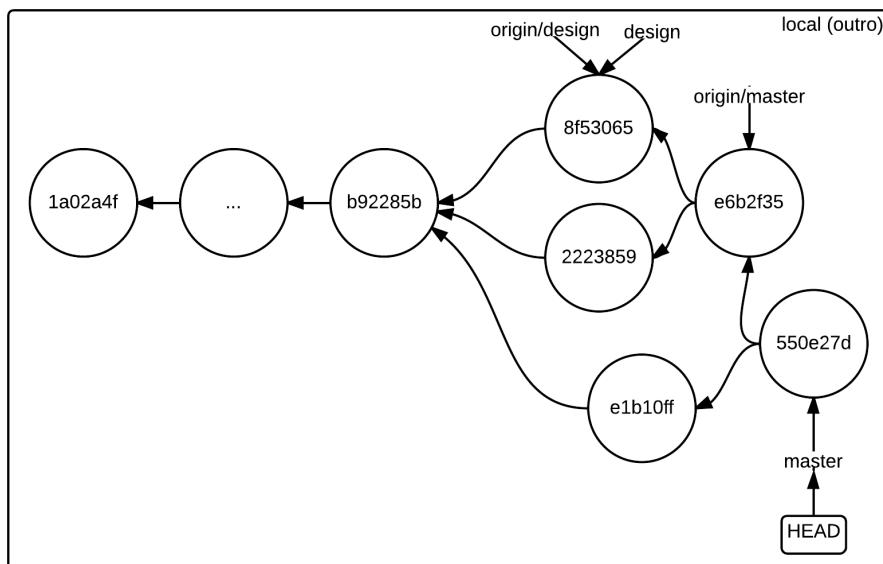



Figura 7.10: Merge de origin/master na branch master do repositório outro

Mesclando branches remotas e locais com rebase

A alternativa ao merge seria um rebase da branch `origin/master` na

branch `master`. Para isso, devemos executar:

```
$ git rebase origin/master
```

Teríamos como resposta:

```
First, rewinding head to replay your work on top of it...
Applying: Adicionando página sobre móveis PET
```

As mudanças lá do GitHub foram aplicadas na branch local `master` do repositório `outro`, mas não houve commit de merge.

O histórico do repositório foi linearizado, o que podemos observar executando `git log -n 5 --oneline --decorate --parents`:

```
0f8d4b3 e6b2f35 (HEAD, master) Adicionando página sobre móveis
                               PET
e6b2f35 2223859 8f53065 (origin/master) Mesclando com a branch
                                        design
2223859 b92285b Alterando textos do banner
8f53065 b92285b (origin/design, design) Adicionando Bootstrap
b92285b a5b9fce Revert "Adicionando texto peculiar"
```

Observe que o commit com mensagem "Adicionando página sobre móveis PET" foi refeito, ficando com o código `0f8d4b3`, após serem aplicados os commits de `origin/master`.

Em geral, não queremos alterar o histórico de commits da branch local `master`. Mas é mais importante ainda evitar mudanças nos commits da branch remota `origin/master`, que é compartilhada por todos os membros da nossa equipe.

Uma alternativa bastante utilizada é não comitar diretamente na branch local `master`, mas em uma outra branch local específica para a funcionalidade sendo desenvolvida. Então, faríamos o rebase da `origin/master` nessa outra branch local, que teria seu histórico modificado. Dessa forma, a `master` ficaria intacta.

Ao final da funcionalidade, a branch local da funcionalidade teria um merge feito na branch local `master`.

O repositório `moveis` e o do GitHub não sofreram mudanças. Visualizando a situação do repositório `outro`, teríamos:

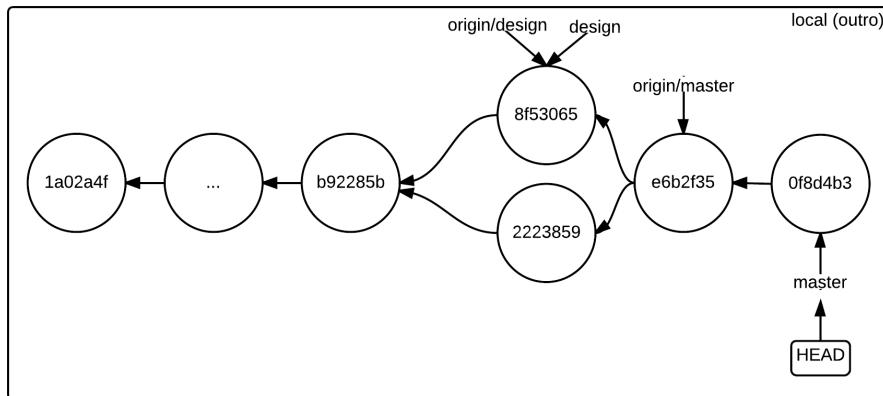



Figura 7.11: Rebase de origin/master na branch master do repositório outro

Obtendo commits e mesclando de uma vez com pull

Para obtermos e mesclarmos os novos commits de uma branch remota com uma branch local poderíamos ter usado um comando só:

```
$ git pull
```

Será aberto um editor de texto para que seja informada a mensagem do commit de merge.

Na saída, teremos algo como:

```
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (5/5), done.
Unpacking objects: 100% (6/6), done.
remote: Total 6 (delta 1), reused 0 (delta 0)
From github.com:fulanodasilva/moveis-ecologicos
   b92285b..e6b2f35  master     -> origin/master
Removing principal.css
Merge made by the 'recursive' strategy.
 index.html      | 29 ++++++++++++++++++++++--------
 js/principal.js |  2 +-
```

```
 principal.css   | 11 -----------
 3 files changed, 22 insertions(+), 20 deletions(-)
 delete mode 100644 principal.css
```

O comando anterior vai ter o mesmo efeito de um `git fetch origin` seguido de um `git merge origin/master`.

Se desejássemos que o `git pull` tivesse um efeito parecido com `git fetch origin` seguido de um `git rebase origin/master`, poderíamos ter passado a opção `--rebase`:

```
$ git pull --rebase
```

Teríamos como resposta:

```
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (6/6), done.
From github.com:fulanodasilva/moveis-ecologicos
   b92285b..e6b2f35  master     -> origin/master
First, rewinding head to replay your work on top of it...
Applying: Adicionando página sobre móveis PET
```

Para saber mais: Qual usar? Um pull ou um pull --rebase?

Usar `git pull` ou `git pull --rebase` ao mesclar com mudanças remotas é uma questão parecida com a de se devemos usar `git merge` ou `git rebase` para mesclagens locais.

O comando `git pull`, não passa de um `git fetch` seguido de um `git merge`. Por isso, temos as mesmas vantagens e desvantagens do merge, mas agora considerando repositórios remotos. Como vantagem, temos um registro fiel do que aconteceu em cada repositório da nossa equipe, com os commits exatos. Mas, como desvantagem, os vários commits de merge complicam o entendimento do histórico do projeto, afetando tarefas como navegar por código antigo e revisar código novo. Em times grandes, um histórico complicado pode tornar o trabalho bastante desafiador.

Na figura a seguir, há um pequeno trecho do histórico de um projeto real em que um dos autores trabalharam, com detalhes omitidos. Fica clara a confusão ocasionada pelos commits de merge:

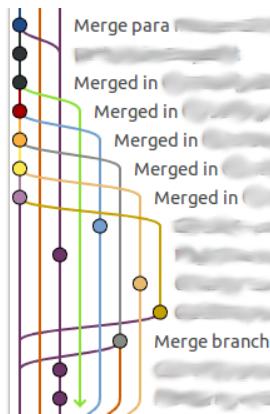



Figura 7.12: Commits de merge em um projeto real

Já o comando `git pull --rebase` é um `git fetch` seguido de um `git rebase`. O histórico é linearizado, simplificando nosso trabalho. Porém, a cada rebase alguns commits são reescritos e, no fim das contas, estamos perdendo informações sobre nosso repositório. Quando acontecem conflitos, corrigi-los pode ser bastante complicado.

O rebase pode ser especialmente problemático se acabarmos mudando um commit já compartilhado com outros membros da nossa equipe. Nossos colegas podem sofrer na hora de obter os commits modificados, já que o Git pode se perder.

Não parece uma boa solução usar sempre o merge ou sempre o rebase. Um meio termo parece algo mais sensato!

Nossa sugestão é a seguinte:

- Utilizar um `git pull --rebase` para obter mudanças remotas. É uma prática segura porque nossos novos commits locais ainda não foram compartilhados, ou seja, outros membros nem sabem da existência desses commits. Por isso, alterá-los não traz grandes problemas.

- Ao trabalharmos em uma nova funcionalidade utilizando uma branch, marcamos sua entrega fazendo um merge. Assim, conseguimos ter uma boa ideia de quando a funcionalidade começou a ser desenvolvida e quando foi reintegrada à branch `master`.

- Se estivermos trabalhando sozinhos na nova funcionalidade, podemos fazer rebases periódicos da `master` na nossa branch para obter código novo, porém deixando o histórico da branch bem limpo.
- É interessante fazer o push da nossa branch solitária em um repositório remoto, para backup. Mas é importante evitar que alguém faça checkout.

Uma discussão detalhada pode ser encontrada em: http://blog.sourcetreeapp.com/2012/08/21/merge-or-rebase/

7.7 Deletando branches remotas

Vamos dizer que nosso cliente solicitou que fizéssemos uma página de contatos que permite ao usuário do site envio de mensagens.

Para implementar essa solicitação, trabalharemos em uma nova branch chamada `contato`. Para criá-la e já começar a trabalhar nela, executamos o comando `git checkout -b contato`. Teríamos como resposta:

```
Switched to a new branch 'contato'
```

Então, vamos iniciar o desenvolvimento criando um arquivo com o nome `contato.html` e com o seguinte código:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Entre em contato conosco</title>
    </head>
    <body>
        <h1>Fale com a gente.</h1>
        <p>Dúvidas, sugestões ou choramingos</p>
        <form>
            <label for="nome">Nome</label>
            <input id="nome">
            <label for="email">Email</label>
            <input id="email">
```

```
            <label for="mensagem">Mensagem</label>
            <textarea id="mensagem"></textarea>
            <input type="submit" value="Enviar">
        </form>
    </body>
</html>
```

Vamos rastrear nosso novo arquivo com `git add contato.html` e comitá-lo com `git commit -m "Iniciando página de contato"`.

Mas surgiu uma nova demanda urgente! Temos que parar de trabalhar na página de contato. Alguém do nosso time disse que iria dar seguimento. Por isso, resolvemos compartilhar nossa branch executando: `git push origin contato`.

Pronto, a branch `contato` foi publicada e tem uma cópia lá no GitHub.

Ao executarmos o comando `git branch -a -v`, teríamos:

```
* contato                3523c62 Iniciando página de contato
  design                 8f53065 Adicionando Bootstrap
  master                 0f8d4b3 Adicionando página sobre móveis
                                   PET
  remotes/origin/contato 3523c62 Iniciando página de contato
  remotes/origin/design  8f53065 Adicionando Bootstrap
  remotes/origin/master  0f8d4b3 Adicionando página sobre móveis
                                   PET
```

Mas nosso cliente, que é um pouco indeciso, voltou atrás e quis simplificar o projeto, parando o desenvolvimento da página de contato e deixando apenas o e-mail na página principal.

Devido a essa mudança nos requisitos, resolvemos remover nossa branch `contato` mudando para a branch `master` do nosso repositório e executando o comando `git branch -D contato`.

Mas quando executamos o comando anterior, removemos apenas a branch local `contato`. Lá no GitHub, a branch `contato` continua intacta. E a nossa branch remota `origin/contato`, que aponta para a branch correspondente do GitHub, também continua no nosso repositório.

Podemos constatar isso executando novamente o comando `git branch -a -v`:

```
design                  8f53065 Adicionando Bootstrap
* master                  0f8d4b3 Adicionando página sobre móveis
                                  PET
  remotes/origin/contato 3523c62 Iniciando página de contato
  remotes/origin/design  8f53065 Adicionando Bootstrap
  remotes/origin/master  0f8d4b3 Adicionando página sobre móveis
                                    PET
```

Para removermos definitivamente a branch remota `origin/contato` e a branch `contato` lá do GitHub, devemos executar:

```
$ git push origin :contato
```

Teríamos na saída:

```
To git@github.com:fulanodasilva/moveis-ecologicos.git
 - [deleted]         contato
```

Pronto! Se olharmos no GitHub, a branch `contato` não aparece mais!

> No caso da branch lá do servidor ter um nome diferente da branch local, ao deletá-la devemos utilizar o nome da branch do servidor.
>
> Por exemplo, se a branch do servidor fosse `contato_do_servidor`, deveríamos executar `git push origin :contato_do_servidor`.

Capítulo 8

Controlando versões do código com tags

Mostramos o novo design para o nosso cliente e foi um sucesso! Nosso cliente pediu que colocássemos no ar a nova versão, o mais rápido possível.

Ao liberarmos essa nova versão do site para implantação em produção, é uma boa prática tirarmos uma *foto* do código nesse momento. Se houver algum erro, saberemos exatamente o código que está sendo usado em produção e será mais fácil caçar a origem da falha.

Em sistemas de controle de versão, essas fotos de um determinado instante de um repositório são chamadas de **tags**.

No Git, uma tag é simplesmente um apontador fixo para um commit específico. Ao contrário de branches, esse apontador não avança com novos commits.

Em geral, criamos tags com nomes como `v1.0`, `v1.1`, `v2.0` e assim por diante. Cada equipe deve definir o seu padrão.

> Os nomes permitidos para tags e branches têm algumas restrições: não podem conter alguns caracteres especiais como `~`, `^` e `:`, além de sequências de caracteres como `..` e `@{`.
>
> Os detalhes podem ser obtidos em: http://git-scm.com/docs/git-check-ref-format.html

8.1 Criando, listando e deletando tags

Para criarmos uma tag `v1.0` no Git, devemos executar o comando:

```
$ git tag v1.0
```

O gráfico do nosso repositório local `outro` depois do comando anterior ficaria parecido com:

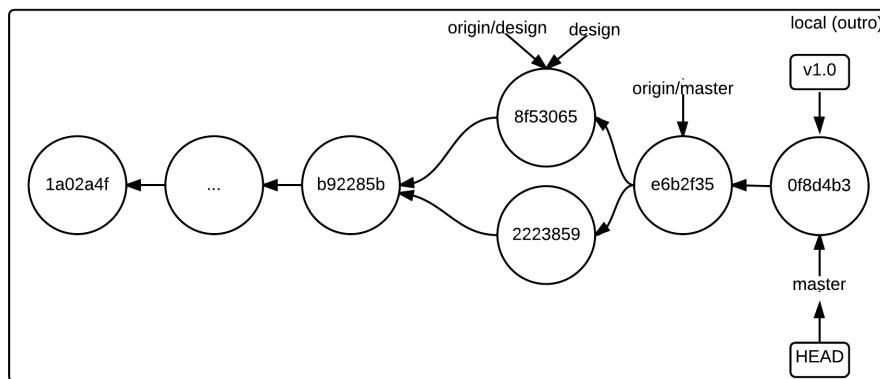



Figura 8.1: Nova tag no repositório outro

Se quisermos listar as tags do nosso repositório, basta executarmos:

```
$ git tag
```

Teríamos como resposta:

```
v1.0
```

Se verificarmos o histórico do repositório com o comando `git log -n 5 --oneline --decorate`, veremos:

```
0f8d4b3 (HEAD, tag: v1.0, master) Adicionando página sobre
                                  móveis PET
e6b2f35 (origin/master) Mesclando com a branch design
2223859 Alterando textos do banner
8f53065 (origin/design, design) Adicionando Bootstrap
b92285b Revert "Adicionando texto peculiar"
```

Observe que o commit `0f8d4b3` possui a tag `v1.0`.

Podemos criar uma tag para um commit passado. Se quisermos criar uma tag chamada `banners` para o commit `2223859` (Alterando textos do banner), devemos executar:

```
$ git tag banners 2223859
```

Imagine que tenhamos errado o nome de uma tag. Por exemplo, criamos a tag `verssao1` com o comando `git tag verssao1`. Podemos deletá-la com o comando:

```
$ git tag -d verssao1
```

Na saída, teremos:

```
Deleted tag 'verssao1' (was 0f8d4b3)
```

8.2 Mais informações com tags anotadas

As tags que acabamos de criar são chamadas de tags leves, porque não passam de um simples apontador fixo para um commit.

Se desejarmos manter mais informações como quando uma tag foi criada, quem a criou, além de termos uma mensagem descritiva, devemos criar **tags anotadas**.

Para criarmos uma tag anotada, basta usarmos a opção `-a` do comando `git tag` e informar uma mensagem com a opção `-m`:

```
$ git tag -a v1.1 -m "Liberando versão urgente"
```

> Se não informarmos uma mensagem ao utilizarmos a opção `-a` do comando `git tag`, será aberto um editor de texto.

Podemos exibir as informações de uma tag anotada com o comando:

```
$ git show -s v1.1
```

Teríamos como resposta detalhes sobre a criação da tag anotada, além de informações sobre o commit para o qual a tag aponta:

```
tag v1.1
Tagger: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Sun Jun 22 13:49:13 2014 -0300

Liberando versão urgente

commit 0f8d4b3f1a5ee4594e72ab1f49786b3a31ab20bb
Author: Fulano da Silva <fulanodasilva.git@gmail.com>
Date:   Fri Jun 13 17:14:16 2014 -0300

Adicionando página sobre móveis PET
```

> O comando `git show` não serve apenas para exibir informações de tags.
>
> É possível utilizá-lo para inspecionar commits e outros detalhes internos do Git.

8.3 Compartilhando tags com a sua equipe

As tags criadas com o comando `git tag`, anotadas ou não, ficam disponíveis apenas no repositório local.

Quando estamos trabalhando em equipe, em geral queremos compartilhar as tags que criamos com nossos colegas, enviando-as para o repositório do servidor. No nosso caso, enviaríamos as tags que criamos para o GitHub.

Para compartilhar tags, fazemos de maneira parecida com o que fizemos com branches, utilizando o comando `git push`. Para compartilhar a tag `v1.0`, devemos executar o comando:

```
$ git push origin v1.0
```

Será exibido algo como:

```
Counting objects: 13, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 1.20 KiB | 0 bytes/s, done.
Total 9 (delta 4), reused 0 (delta 0)
To git@github.com:fulanodasilva/moveis.git
 * [new tag]         v1.0 -> v1.0
```

Pronto! A tag `v1.0` foi enviada para o repositório remoto `origin`. Nossos colegas de equipe poderão obter essa tag quando executarem o comando `git pull`.

Mas ainda temos outras tags para compartilhar: as tags `banners` e `v1.1`. Executar o comando `git push` para cada tag é algo bastante tedioso.

Se quisermos enviar todas as novas tags do repositório local para um repositório remoto podemos utilizar o comando `git push` com a opção `--tags`:

```
$ git push origin --tags
```

Na saída, teremos:

```
Counting objects: 1, done.
Writing objects: 100% (1/1), 181 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:fulanodasilva/moveis.git
 * [new tag]         banners -> banners
 * [new tag]         v1.1 -> v1.1
```

Se olharmos no GitHub, veremos que nossas tags estão lá!

Capítulo 9

Lidando com conflitos

Ao mesclarmos mudanças entre duas branches locais no capítulo 6 e entre uma branch local e uma remota no capítulo 7, o Git soube fazer essa mesclagem automaticamente, sem qualquer problema.

Comparado com outros sistemas de controles de versão, o Git é especialmente bom ao juntar mudanças de duas branches diferentes.

Nos capítulos anteriores, fizemos mudanças em arquivos diferentes, o que o Git resolve com facilidade. Até mudanças em um mesmo arquivo podem ser resolvidas.

Mas mesmo o Git não consegue fazer milagres!

9.1 Mesclando mudanças em um mesmo arquivo sem conflitos

Nosso cliente ligou desesperado dizendo que descobriu um erro importante no site: estava faltando o acrônimo "S. A." no título da página.

Precisando atendê-lo com urgência, fizemos a seguinte modificação na branch `master`, no arquivo `index.html`:

```
<!-- início do arquivo ... -->
    <head>
        <!-- tags meta e link ... -->
        <title>Móveis Ecológicos S. A.</title>
        <!-- tag script -->
    </head>
<!-- restante do arquivo ... -->
```

Com a alteração feita, comitamos o arquivo `index.html` com o comando `git commit -am "Corrigindo título da página"`. Obtivemos na saída:

```
[master 1963258] Corrigindo título da página
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Outra coisa que nosso cliente nos informou é que a página da Móveis Ecológicos S. A. estava sendo copiada por outras empresas do ramo.

Sugerimos algumas soluções para minimizar o problema mas nosso cliente decidiu por uma maneira simples: colocar um rodapé na página com o símbolo de *copyright*. Mas o rodapé teria que ser bonito, "o rodapé mais belo de todos os tempos", segundo nosso cliente.

Para podermos trabalhar com calma em busca do rodapé perfeito, passando a trabalhar na branch `design` depois de executar o comando `git checkout design`.

Depois de algum esforço chegamos a um design utilizando classes do framework CSS Bootstrap. Modificamos o arquivo `index.html` conforme a seguir:

```
<!-- inicio do arquivo -->
            <p class="text-muted text-right">
```

```
                &copy; Copyright Móveis Ecológicos S. A.
            </p>
        <div>
    </body>
</html>
```

Então, adicionamos a mudança e a gravamos no repositório com o comando `git commit -am "Adicionando rodapé com copyright"`. Na saída, foi mostrado:

```
[design 0e2d613] Adicionando rodapé com copyright
 1 file changed, 3 insertions(+)
```

Listando o histórico dos últimos dois commits com o comando `git log -n 2 --oneline --decorate --all`, teríamos:

```
0e2d613 (HEAD, design) Adicionando rodapé com copyright
1963258 (master) Corrigindo título da página
```

O commit de código `0e2d613` (Adicionando rodapé com copyright) ainda não foi mesclado na `master`.

Vamos voltar para a branch `master` executando o comando `git checkout master`.

Agora, vamos fazer o merge da branch `design` na `master` com o comando `git merge design -m "Merge de design na master"`. Teremos como resposta:

```
Auto-merging index.html
Merge made by the 'recursive' strategy.
 index.html | 3 +++
 1 file changed, 3 insertions(+)
```

Se observarmos o conteúdo do arquivo `index.html`, o conteúdo está de acordo com as últimas alterações: o título da página foi atualizado e foi inserido um rodapé com informações de copyright.

O Git conseguiu fazer o merge automático, mesclando sem problemas alterações no arquivo `index.html` das branches `design` e `master`. Isso aconteceu porque alteramos o arquivo em **áreas distintas**.

> Se tivéssemos feito um `git rebase design` para mesclar as alterações da branch `design` na `master`, também não teríamos nenhum conflito.
>
> No caso das alterações serem feitas em uma branch remota, poderíamos utilizar um `git pull` ou um `git pull --rebase` sem nenhum conflito.

9.2 Conflitos após um merge com mudanças em um mesmo arquivo

Deixamos nosso cliente satisfeito. E com a satisfação surgiram novas ideias!

Uma das coisas que nosso cliente pediu foi colocar um texto bastante criativo no topo da página. E solicitou que a mudança fosse urgente!

Por isso, na branch `master`, editamos o arquivo `index.html` inserindo logo abaixo da tag `h2` o seguinte parágrafo:

```
<!-- inicio do arquivo até tag h2-->
<p>Do lixo ao luxo, um resgate do descartado e uma transformação
em algo funcional. Reaproveitar é aproveitar. Coisas descartadas
são nossa carta na manga. Erodido, mas único.</p>
<!-- ul e restante do arquivo -->
```

Na branch `master`, comitamos as alterações no `index.html` executando o comando `git commit -am "Adicionando texto criativo"`. Temos como resposta:

```
[master d2f3597] Adicionando texto criativo
 1 file changed, 3 insertions(+)
```

Outro pedido do nosso cliente foi melhorar o design do banner, de maneira a destacá-los.

Para trabalhar nessa demanda, mudamos para a branch `design` executando o comando `git checkout design`.

Pesquisamos um pouco e resolvemos utilizar um recurso do Bootstrap para embelezar o banner. Na branch `design`, editamos o arquivo `index.html`, inserindo as seguintes classes CSS na tag `h2`:

```
<!-- inicio do arquivo até tag h1-->
    <h2 id="mensagem" class="alert alert-info"></h2>
<!-- ul e restante do arquivo -->
```

Depois disso, na branch `design`, comitamos as alterações no arquivo `index.html` executando o comando `git commit -am "Melhorando design do banner"`. Na saída, tivemos:

```
[design fa528a5] Melhorando design do banner
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Ainda na branch `design`, ao exibirmos o histórico dos últimos dois commits do repositório com o comando `git log -n 2 --oneline --decorate --all`, teríamos:

```
fa528a5 (HEAD, design) Melhorando design do banner
d2f3597 (master) Adicionando texto criativo
```

Note que o commit de código `fa528a5` (Melhorando design do banner) ainda não teve suas alterações aplicadas na branch `master`.

Executando o comando `git checkout master`, voltamos a trabalhar na branch `master`.

Para mesclarmos o último commit da branch `design` na `master`, executamos o comando `git merge design -m "Merge de design na master"`. Na saída, teremos:

```
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed;fix conflicts and then commit the result.
```

Observe a mensagem anterior: foi indicado que houve um conflito no arquivo `index.html` e que o merge automático falhou.

Se abrirmos o arquivo `index.html` veremos um conteúdo estranho:

```
<!-- inicio do arquivo arquivo até h1 -->
<<<<<<< HEAD
<h2 id="mensagem"></h2>
<p>Do lixo ao luxo, um resgate do descartado e uma transformação
```

```
em algo funcional. Reaproveitar é aproveitar. Coisas descartadas
são nossa carta na manga. Erodido, mas único.</p>
=======
<h2 id="mensagem" class="alert alert-info"></h2>
>>>>>>> design
<!-- ul e restante do arquivo -->
```

Mesmo sendo muito bom para fazer o merge automático, dessa vez o Git não conseguiu mesclar o arquivo `index.html` de maneira clara porque foram feitas alterações na **mesma região do arquivo**.

De forma cautelosa, o Git marcou um conflito com os caracteres `<`, `=` e `>`.

Entre os textos `<<<<<<< HEAD` e `=======` estão as alterações que fizemos na branch `master`, que é a branch atual, para qual o `HEAD` está apontando.

Já entre `=======` e `>>>>>>> design`, estão as alterações que fizemos na branch `design`.

Ao executarmos `git status`, teremos:

```
On branch master
Your branch is ahead of 'origin/master' by 5 commits.
  (use "git push" to publish your local commits)

You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or
"git commit -a")
```

Note que o arquivo `index.html` aparece como não mesclado, como *both modified* sob *Unmerged paths* (nomes não muito intuitivos).

Agora teremos que realizar o merge manualmente. Nesse caso, a solução é fácil. Vamos editar o arquivo `index.html` da seguinte maneira:

```
<!-- inicio do arquivo arquivo até h1 -->
<h2 id="mensagem" class="alert alert-info"></h2>
<p>Do lixo ao luxo, um resgate do descartado e uma transformação
em algo funcional. Reaproveitar é aproveitar. Coisas descartadas
são nossa carta na manga. Erodido, mas único.</p>
<!-- ul e restante do arquivo -->
```

Mesmo depois de editar o arquivo, removendo as marcações de conflito e mesclando manualmente as alterações, ainda teremos a mesma resposta de antes ao executarmos `git status`.

Para informarmos para o Git que resolvemos o conflito no arquivo, temos que adicioná-lo à área de stage executando `git add index.html`.

Agora, ao executarmos `git status`, teremos:

```
On branch master
Your branch is ahead of 'origin/master' by 5 commits.
  (use "git push" to publish your local commits)

All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html
```

Repare que o arquivo `index.html` aparece como modificado e está pronto para ser comitado.

Depois disso, falta comitarmos as mudanças com o comando `git commit -am "Resolvendo conflitos após merge de design"`. Como resposta, veremos:

```
[master d364c56] Resolvendo conflitos após merge de design
```

Pronto! Conflito resolvido. Finalmente o merge foi finalizado.

Nossa página ficou com um visual bem razoável:

Figura 9.1: Página embelezada

9.3 Resolvendo conflitos após um rebase

E se tivéssemos utilizado um rebase para mesclar as últimas mudanças da branch `design` na `master`?

Devemos unir o commit `fa528a5` (Melhorando design do banner) da branch `design` com o commit `d2f3597` (Adicionando texto criativo) da branch `master`. Mas ambos alteram o topo do arquivo `index.html`, o que gera um conflito.

Considerando que estamos na branch `master`, após executar o comando `git rebase design`, veríamos:

```
First, rewinding head to replay your work on top of it...
Applying: Alterando textos do banner
Applying: Adicionando página sobre móveis PET
Applying: Corrigindo título da página
Applying: Adicionando texto criativo
Using index info to reconstruct a base tree...
M	index.html
Falling back to patching base and 3-way merge...
```

```
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Failed to merge in the changes.
Patch failed at 0004 Adicionando texto criativo
The copy of the patch that failed is found in:
   /home/fulanodasilva/moveis/.git/rebase-apply/patch

When you have resolved this problem, run "git rebase
--continue".
If you prefer to skip this patch, run "git rebase --skip"
instead.
To check out the original branch and stop rebasing, run
"git rebase --abort".
```

Observe que que aparece um conflito no arquivo `index.html`.

Ao executarmos o comando `git status`, teríamos:

```
rebase in progress; onto fa528a5
You are currently rebasing branch 'master' on 'fa528a5'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git reset HEAD <file>..." to unstage)
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

De maneira semelhante ao que aconteceu após o conflito no merge, o arquivo `index.html` aparece em *Unmerged paths* como *both modified*. Apesar da descrição pouco intuitiva, isso indica que há um conflito nesse arquivo.

Ao abrirmos o arquivo `index.html` veremos as mesmas marcações de conflito com os textos `<<<<<<< HEAD`, `=======` e `>>>>>>> design`.

Devemos mesclar o arquivo `index.html` manualmente, removendo as marcações de conflito, conforme fizemos anteriormente.

Para marcar a resolução do conflito, devemos executar `git add index.html`.

Depois disso, ao executar `git status`, teríamos:

```
rebase in progress; onto fa528a5
You are currently rebasing branch 'master' on 'fa528a5'.
  (all conflicts fixed: run "git rebase --continue")

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html
```

Para finalizar o rebase, teríamos que executar ainda:

```
$ git rebase --continue
```

Pronto! Finalmente finalizamos nosso rebase conflitante! Ufa!

> Se quiséssemos abortar o rebase, voltando à situação antes da tentativa de mesclagem, poderíamos executar `git rebase --abort`.
>
> Também poderíamos simplesmente ignorar o commit que gerou o conflito, pulando-o, executando `git rebase --skip`. Essa opção tem que ser usada com cuidado. Só faz sentido quando exatamente a mesma alteração foi feita na branch sendo mesclada.

9.4 Usando uma ferramenta para resolver conflitos

O conflito que acabamos de resolver foi bem fácil. Visualizar o arquivo para decidir como mesclar as diferentes versões não demandou grande esforço.

Porém, algumas vezes temos conflitos bem desafiadores, em que é difícil visualizar cada trecho conflitante do arquivo.

Além disso, podemos ter vários arquivos com conflito para serem solucionados.

Experiência no código que está sendo desenvolvido e calma são fundamentais para não inserirmos erros ao resolver conflitos.

Mas utilizar uma ferramenta apropriada pode auxiliar na rapidez de resolução de conflitos.

Com Git, podemos invocar uma ferramenta de resolução de conflitos com o comando:

```
$ git mergetool
```

Seriam exibidas mensagens parecidas com:

```
This message is displayed because 'merge.tool' is not
configured.
See 'git mergetool --tool-help' or 'git help config' for more
details. 'git mergetool' will now attempt to use one of the
following tools: meld opendiff kdiff3 tkdiff xxdiff
tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge araxis
bc3 codecompare emerge vimdiff
Merging:
index.html

Normal merge conflict for 'index.html':
  {local}: modified file
  {remote}: modified file
Hit return to start merge resolution tool (meld):
```

Leia com atenção a mensagem anterior. Repare que o Git listou algumas ferramentas para resolução de conflitos como *Meld* (http://meldmerge.org) , bem simples e efetiva que funciona em Windows, Mac OS e Linux, TortoiseMerge (http://tortoisesvn.net/TortoiseMerge.html) , bastante utilizada mas só para Windows, entre outras.

Alguma das ferramentas listadas anteriormente será procurada e invocada.

Na mensagem anterior, foi indicada a ferramenta Meld. Pressionando a tecla `Enter`, a ferramenta é aberta:

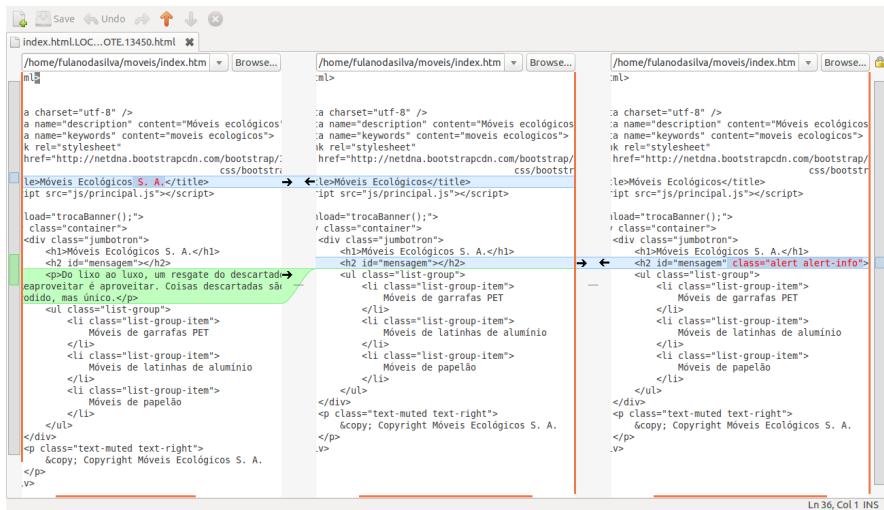



Figura 9.2: Meld, ferramenta de resolução de conflitos

> Para definir uma ferramenta padrão para resolução de conflitos podemos utilizar a configuração `merge.tool`.
>
> Por exemplo, para sempre utilizar o Meld, poderíamos executar `git config --global merge.tool meld`. O comando `meld` deve estar no PATH do sistema operacional.

No Meld, o arquivo `index.html` foi aberto à esquerda com o conteúdo da `master`, à direita com o conteúdo da branch `design` e no centro com o conteúdo anterior às duas modificações conflitantes. Devemos editar o arquivo do centro

Depois de resolvido o conflito usando o Meld e salvar o arquivo, se executarmos `git status`, teremos:

```
On branch master
Your branch is ahead of 'origin/master' by 5 commits.
  (use "git push" to publish your local commits)

All conflicts fixed but you are still merging.
```

```
  (use "git commit" to conclude merge)

Changes to be committed:

    modified:   index.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    index.html.orig
```

Note que o arquivo `index.html` está entre as mudanças prontas para serem comitadas. Além disso, foi criado um novo arquivo chamado `index.html.orig` com o conteúdo original, antes do conflito ser resolvido. O arquivo `index.html.orig` pode ser apagado.

Podemos, então, comitar o arquivo `index.html` agora sem conflitos, executando `git commit -am "Resolvendo conflitos após merge de design"`. Na saída, teremos:

```
[master b0418dc] Resolvendo conflitos após merge de design
```

Capítulo 10

Maneiras de trabalhar com Git

No capítulo 3, vimos como trabalhar com um repositório local do Git, criando uma "máquina do tempo" para o nosso código.

Já no capítulo 4, vimos como utilizar repositórios remotos para compartilhar nosso código com os outros membros da nossa equipe, além de manter um backup do repositório.

E no capítulo 5, utilizamos o Github para tornar nosso repositório remoto acessível de qualquer lugar que tenha acesso à internet.

Depois, nos capítulos 6 e 7, utilizamos branches locais e remotas para trabalhar em paralelo em diferentes funcionalidades de maneira organizada.

Estudamos também, no capítulo 8, como marcar as versões de nossas entregas através de tags.

No capítulo 9 aprendemos como lidar com conflitos que acontecem ao mesclar branches.

Mas como usar tudo isso que aprendemos no seu projeto? Qual a melhor maneira trabalhar com o Git no seu caso?

Modelos de distribuição de repositórios e de branches

O Git é um sistema de controle de versão distribuído. Temos localmente uma cópia completa do repositório, que podemos compartilhar remotamente. Além disso, podemos apontar para mais de um repositório remoto.

Por isso, uma das decisões que precisamos fazer é o **modelo de distribuição de repositórios** a ser adotado, definindo como organizar os repositórios remotos. Os modelos mais utilizados são:

- Apenas um repositório remoto, central, para onde os repositórios locais apontam;
- Cada desenvolvedor tem seu fork, um repositório remoto que é uma cópia do projeto, utilizando um repositório central para integração;
- Uma hierarquia de repositórios para integração.

Ao contrário de outros sistemas de controle de versão, criar branches no Git é algo muito leve e rápido. Além disso, o Git é muito bom em fazer mesclagens automáticas, o que encoraja o uso de branches.

Portanto, outra coisa que precisamos decidir é o **modelo de branches** a ser utilizado no nosso projeto. Alguns dos modelos mais comuns são:

- Utilizar apenas a branch `master`;
- Ter uma branch para cada nova funcionalidade, deixando a `master` para código pronto para ser entregue;
- Ter algumas branches por etapa de desenvolvimento, como uma branch de longo prazo para código ainda em construção e uma de curto prazo para correções de bugs urgentes.

Existem vários fluxos de trabalho possíveis ao se trabalhar com Git. Vamos estudar os mais comuns.

10.1 Utilizando só a branch master com um repositório central

Um dos fluxos de trabalho mais simples é utilizar apenas um repositório central hospedado, por exemplo, no Github, comitando tudo diretamente na branch local `master`.

Para enviar os commits locais, cada desenvolvedor faz um push para o repositório central. Por isso, todos os membros da equipe devem ter permissões de push.

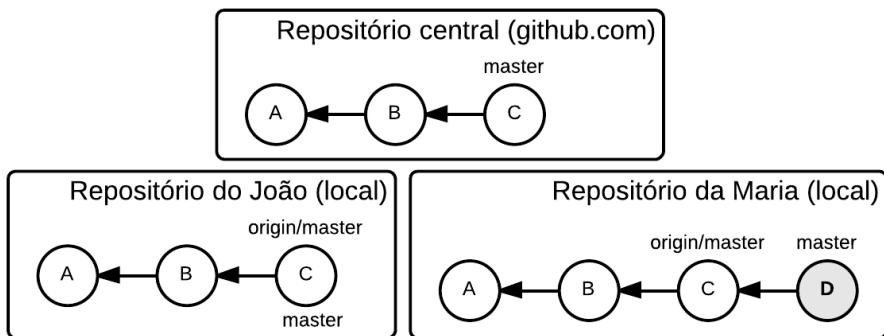



Figura 10.1: Repositório central só com master

Considerando que temos um repositório para o nosso projeto configurado no Github, a primeira coisa seria fazer:

```
$ git clone https://github.com/empresa/projeto.git
```

Após o clone, temos uma cópia local completa do repositório.

Podemos, então, criar novos arquivos ou editar arquivos existentes. Quando estivermos satisfeitos com o código, podemos adicionar as alterações à área de stage e depois comitá-las na `master`, utilizando comandos como:

```
$ git add .
$ git commit -m "Otimizando consulta a clientes"
```

Depois de alguns commits, podemos compartilhar nosso trabalho com a equipe:

```
$ git push origin master
```

Depois de termos publicado os novos commits, os outros desenvolvedores podem obtê-los executando:

```
$ git pull --rebase origin master
```

> Encorajamos o uso de um `git pull --rebase` ao obter mudanças remotas porque simplifica o histórico do repositório, evitando commits de merge.

Se ocorrer algum conflito no momento em que obtemos os novos commits, devemos resolver os problemas editando os arquivos apropriados para então executar:

```
$ git add .
$ git rebase --continue
```

Quando estivermos prontos para fazer uma entrega, podemos marcá-la criando uma tag e, depois, enviando essa tag para o repositório central:

```
$ git tag v1.0
$ git push origin --tags
```

Quando utilizar?

Para equipes pequenas ou na adoção do Git.

Quando trabalhamos em equipes pequenas, com até 5 desenvolvedores, quanto mais simples o fluxo de trabalho, melhor. Como a equipe é pequena, muito provavelmente o projeto será pequeno e teremos controle no ritmo das entregas, não precisando de um fluxo mais poderoso (e complexo).

Esse fluxo é parecido com a maneira como são comumente utilizados sistemas de controle de versão centralizados, como o Subversion. Por isso, para

equipes que estão iniciando com o Git, é o fluxo de mais fácil adoção. A equipe pode focar em aprender os comandos básicos do Git, utilizados para lidar com um repositório local, para depois partir para fluxos mais avançados.

Vantagens

- A simplicidade desse fluxo permite uma adoção mais tranquila para quem está começando a utilizar o fluxo. Também há menos complicação para equipes reduzidas.
- É mais fácil de adotar integração contínua, uma prática bastante comum em projetos que usam metodologias ágeis. Nessa prática, o código deve ser integrado frequentemente, disparando *builds* e testes automatizados e detectando erros de integração o mais rápido possível. Nesse fluxo, como toda alteração é comitada na branch `master`, todo código compartilhado depois de um push será integrado na `master` do repositório central. A cada novo push, o build e os testes automatizados podem ser disparados utilizando esse código integrado. Além disso, possíveis conflitos são detectados a cada pull.

Desvantagens

- Ao corrigir defeitos urgentes, pode ser que o código de novas funcionalidades ainda em desenvolvimento já tenha sido compartilhado na branch `master` do repositório central. Com esse fluxo, fica difícil separar o código da correção do defeito do código das novas funcionalidades, o que pode gerar mais defeitos e insatisfação nos clientes.
- Como tudo é comitado na branch `master`, as entregas são feitas com todo o código que está no repositório no momento da entrega. Não é possível entregar só parte das funcionalidades que foram compartilhadas com o repositório central.
- Há a necessidade de permissão de push para todos os membros da equipe no repositório central, já que há apenas um repositório remoto.

Para projetos open source, é algo inviável. Para projetos e equipes muito grandes, pode também ser um problema.

10.2 Utilizando branches por funcionalidade com um repositório central

Podemos utilizar branches para isolar o código de novas funcionalidades ou alterações em funcionalidades existentes. A branch `master` passa a ser tratada de maneira especial, ficando estável durante todo o desenvolvimento do projeto. Só quando a funcionalidade estiver pronta, é feito um merge da branch da funcionalidade na `master`.

Se for preciso realizar uma correção urgente, pode ser utilizada a branch `master` diretamente. Para correções ou alterações mais demoradas, podem ser criadas branches para segregar os commits da correção.

Essa maneira de trabalhar também é chamada de `feature branching` ou `topic branching`.

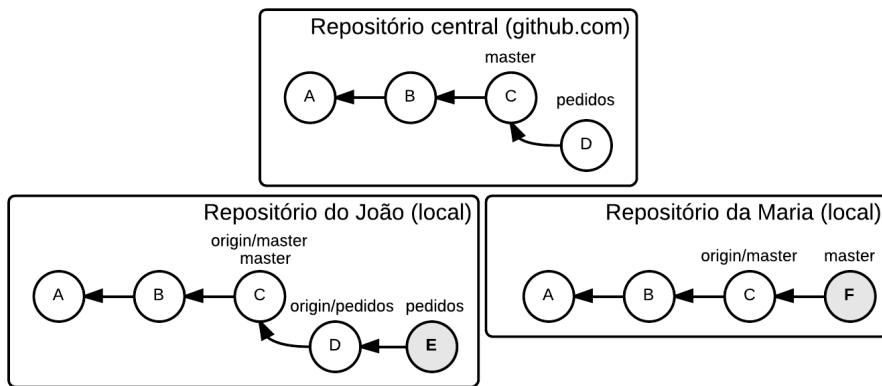



Figura 10.2: Repositório central com branches por funcionalidade

Vamos dizer que temos um repositório local clonado a partir de um repositório do Github, como fizemos anteriormente, com o comando `git clone` https://github.com/empresa/projeto.git .

A funcionalidade a ser desenvolvida é a criação de pedidos de nossa loja

online. Antes de começar, vamos criar a partir da `master` uma branch chamada `pedidos`, já mudando pra essa nova branch:

```
$ git checkout -b pedidos
```

Já na branch `pedidos`, podemos começar a trabalhar na funcionalidade, criando novos arquivos e alterando arquivos existentes. Em seguida, podemos efetuar commits com os comandos que já conhecemos:

```
$ git add .
$ git commit -m "Tela inicial de pedidos"
```

Depois de alguns commits, mesmo sem termos terminado a funcionalidade de pedidos, podemos compartilhar o código que fizemos enviando-o para o repositório central. Basta executar:

```
$ git push origin pedidos
```

Com o código no repositório central, temos um backup do código e outros membros da nossa equipe podem obtê-lo para colaborar no desenvolvimento da funcionalidade. Tudo isso sem afetar a estabilidade da branch `master`.

O restante da nossa equipe, para poder trabalhar na funcionalidade, precisa obter as últimas alterações do repositório central e, em seguida, criar uma cópia local da branch `pedidos`. Para isso, deve-se executar:

```
$ git fetch origin
$ git checkout -t origin/pedidos
```

Nós e outros membros da equipe podemos fazer commits e, quando apropriado, compartilhar o código por meio do repositório central através do comando:

```
$ git push origin pedidos
```

Para obtermos o novo código da funcionalidade, ainda na branch `pedidos`, devemos fazer:

```
$ git pull --rebase origin pedidos
```

No caso de conflitos após o pull com rebase, devemos mesclar os arquivos manualmente e executar:

```
$ git add .
$ git rebase --continue
```

A branch `pedidos` do repositório central terá o código que integra as mudanças de todos os membros da equipe. Testadores e outros interessados podem utilizar o código da branch `pedidos` para verificar o andamento das atividades e detectar problemas. Revisar a qualidade do código, por exemplo, fica mais fácil porque as mudanças relacionadas à funcionalidade ficam isoladas.

Caso haja alguma alteração feita diretamente na `master`, como uma correção de um defeito urgente, é importante obtê-la no repositório local. Para isso, continuando na branch `pedidos`, devemos executar:

```
$ git pull --rebase origin master
```

> O comando anterior obtém as mudanças da branch remota `origin/master` e já faz o rebase na branch atual, que é a `pedidos`. Seria equivalente à sequência de comandos:
>
> ```
> $ git checkout master
> $ git pull origin master
> $ git checkout pedidos
> $ git rebase master
> ```
>
> A diferença é que, nessa última sequência, a branch local `master` ficaria atualizada com as últimas mudanças da `origin/master`.

Quando a equipe estiver satisfeita com a funcionalidade e com o código, algum dos desenvolvedores pode fazer o merge na branch `master`.

Antes, é importante nos certificarmos de que a branch `master` contém as últimas modificações do repositório central. Podemos utilizar um pull sem rebase, para deixarmos a `master` intacta. É tranquilo fazer o pull sem rebase

nesse caso porque, muito provavelmente, será feito um merge do estilo fast-forward, já que não é comum fazermos commits diretamente na `master`.

Na branch `pedidos`, devemos executar os comandos:

```
$ git checkout master
$ git pull origin master
$ git merge pedidos
```

> Há casos em que a equipe tem muitos membros e/ou tem problemas na comunicação. Em vez do merge, talvez seja melhor fazer um `git pull origin pedidos` (ainda na branch `master`). Também seria realizado um merge, mas com possíveis alterações de última hora.

No caso de conflitos no merge da branch da funcionalidade, devemos resolver os conflitos manualmente para depois executar:

```
$ git add .
$ git commit -m "Resolvendo conflitos no merge de pedidos"
```

Depois de realizado o merge da branch `pedidos` com a branch local `master`, podemos compartilhar o código executando:

```
$ git push origin master
```

No momento adequado, podemos liberar uma nova versão do sistema, marcando a entrega com uma tag através do comando `git tag v1.0` e compartilhando a tag criada com o comando `$ git push origin --tags`.

Se não quisermos entregar alguma funcionalidade na nova versão do sistema, basta não fazermos o merge da branch da funcionalidade na branch `master`.

Quando utilizar?

Em projetos um pouco maiores, principalmente se já tiverem algumas entregas feitas. É importante que a equipe tenha familiaridade no uso do Git.

Para projetos que já estão a todo vapor, com melhorias e correções que precisam ser feitas de imediato, isolar código que ainda está sendo desenvolvimento é algo importante. Com o uso de uma branch para cada funcionalidade, podemos organizar o desenvolvimento das novas funcionalidades de maneira a não afetar demandas urgentes.

Para que essa fluxo seja usado com fluência, é preciso que os desenvolvedores já estejam confortáveis com o uso básico do Git.

Vantagens

- Podemos isolar código mais estável na branch `master`, facilitando a realização de melhorias e correções imediatas.
- Revisões da qualidade do novo código que implementa uma funcionalidade podem ser feitas analisando os commits da branch da funcionalidade.
- Pode ser entregue apenas parte das funcionalidades que estão sendo desenvolvidas, possibilitando mudanças mais tranquilas na estratégia de negócio do nosso cliente.

Desvantagens

- Como a equipe precisa dominar o Git razoavelmente bem, o uso desse fluxo no início da adoção do Git fica dificultado.
- Como trabalhamos com um repositório central, ainda há a necessidade de permissão de push para todos os membros da equipe. Por isso, esse fluxo pode ser problemático para grandes equipes e torna-se inviável para projetos open source.
- O código de uma funcionalidade só é efetivamente integrado com outras mudanças no momento do merge final com a branch `master`.

Com outras funcionalidades em desenvolvimento, possíveis conflitos entre o código das funcionalidades só serão descobertos tardiamente, ao mesclarmos todas as branches.

- Realizar integração contínua, descobrindo problemas no código e nas funcionalidades rapidamente, fica mais difícil. A grande vantagem desse fluxo, que é a de isolar código das funcionalidades, torna-se a pior desvantagem sob a ótica de integração contínua. Por isso, especialistas em integração contínua não recomendam o uso de branches por funcionalidade, favorecendo o isolamento das funcionalidades com uma arquitetura mais modular, utilizando abstrações para grandes alterações no código (*branch by abstraction*) e configurações para desabilitar funcionalidades novas (*feature toggles*).

10.3 Utilizando branches por etapa de desenvolvimento com um repositório central

Ao comitarmos um novo código em uma branch separada para uma funcionalidade específica, mantemos a branch local `master` estável, mas corremos o risco de adiar demais a integração entre os códigos das novas funcionalidades. Essa demora pode levar a defeitos que deixariam nossos clientes insatisfeitos.

Para evitar a integração tardia das funcionalidades, poderíamos ter uma *branch para código ainda em desenvolvimento*, chamada `desenv`. Seria uma branch de longo prazo, que existiria enquanto o projeto estiver sendo desenvolvido. A branch `master` teria código já pronto pra ser entregue. Já a branch `desenv` teria código para a próxima entrega. Quando tivermos um ponto estável no novo código, faríamos um merge da `desenv` na `master`.

Podemos continuar utilizando *branches por funcionalidade* mas, agora, criando-as a partir da branch `desenv`. Periodicamente, poderíamos efetuar um merge precoce das branches das funcionalidades, mesmo antes de o código estar totalmente finalizado. Assim, evitaríamos a integração tardia e deixaríamos de afetar a branch `master`.

Seria interessante termos *branches de release*, para comitarmos código referente a uma determinada entrega, como correções de última hora de pe-

quenos bugs descobertos logo antes de liberar uma versão. Teriam nomes como `release1.1` ou `release2.0` e seriam criadas a partir da `desenv`. Também poderiam ser comitados nessas branches de release códigos necessários para preparar uma entrega, como arquivos de versão e *release notes*. Seriam branches de curto prazo, que poderiam ser deletadas quando não fizerem mais sentido. Os commits das correções feitas nessas branches de release precisariam ser aplicados na `master` e na `desenv`, através de um merge.

Para bugs urgentes, que afetam versões em produção, poderíamos criar uma *branch de hotfix*. Se estivermos com a versão `1.0` em produção e acontecer um defeito que pode ser corrigido de maneira imediata, poderíamos criar a branch `hotfix-1.0.1` a partir da `master`. A correção seria feita nessa nova branch e, quando finalizada, faríamos um merge na branch `master`. Também é importante efetuarmos um merge na branch `desenv`, para obtermos a correção do defeito. Uma branch de hotfix é de curto prazo e, depois de feito o merge, podemos apagá-la.

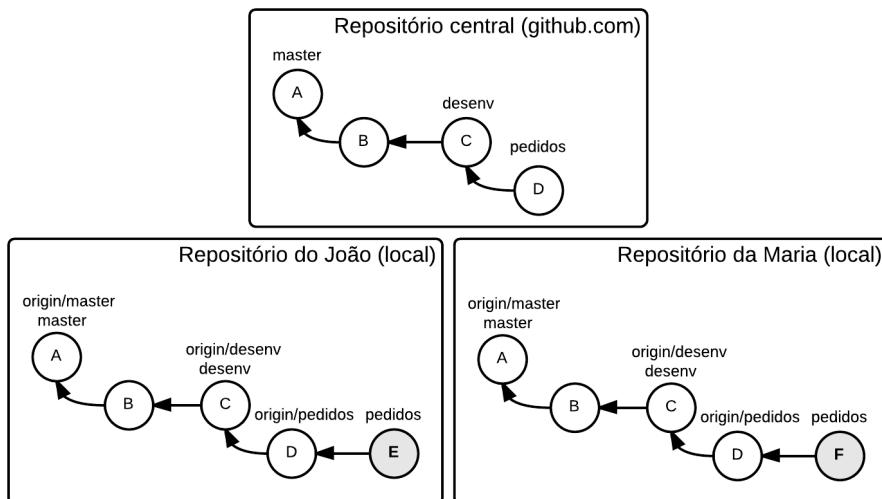



Figura 10.3: Repositório central com branches por etapa de desenvolvimento

Como nos fluxos de trabalho anteriores, cada desenvolvedor deve fazer o clone do repositório central executando, por exemplo, `git clone` https:

//github.com/empresa/projeto.git .

Logo no início do projeto, ou em algum outro momento apropriado, podemos criar uma branch local chamada desenv, a partir da master, para comitarmos código ainda em desenvolvimento. Também é importante que essa branch exista no repositório central. Para isso, um dos desenvolvedores deve executar:

```
$ git branch desenv
$ git push origin desenv
```

Para começarem a trabalhar na branch desenv, os desenvolvedores precisam criar uma branch local que aponta para a branch remota origin/desenv do repositório central. Para isso, devem executar:

```
$ git fetch origin
$ git checkout -t origin/desenv
```

Para trabalharmos em uma nova funcionalidade, por exemplo de estoque, podemos criar uma branch para essa funcionalidade a partir da branch desenv, executando:

```
$ git checkout -b estoque desenv
```

Podemos efetuar alguns commits e, depois, compartilhar a branch com a nossa equipe, através do comando git push origin estoque.

No momento adequado, devemos fazer o merge da funcionalidade estoque na branch desenv, integrando-a com as outras funcionalidades que já foram mescladas com desenv. Porém, antes do merge, é importante obtermos as últimas mudanças em desenv feitas pelos outros membros da nossa equipe através de um pull de origin/desenv. Depois do merge, podemos compartilhar a branch desenv que já terá os últimos commits da estoque. Para fazer isso tudo, considerando que estamos na branch estoque, devemos executar:

```
$ git pull origin desenv
$ git checkout desenv
$ git merge estoque
$ git push origin desenv
```

Podemos repetir os comandos anteriores sempre que quisermos integrar a branch da funcionalidade `estoque` com a branch `desenv`. Ao terminarmos a funcionalidade, devemos fazer um último merge em `desenv`. Se quisermos, podemos deletar nossa branch local `estoque` com o comando `git branch -d estoque`.

Quando estivermos satisfeitos com as funcionalidades que foram mescladas na branch `desenv`, podemos criar uma branch para a próxima release com um nome como `release-1.0` a partir da `desenv`. Para isso, devemos executar:

```
$ git checkout -b release-1.0 desenv
```

Na branch `release-1.0` podemos comitar, por exemplo, um arquivo listando as novas funcionalidades (o release notes), modificar arquivos para preparar a entrega. Além disso, podemos comitar correções de pequenos bugs que descobrirmos antes de liberar a nova versão. Enquanto isso, outros desenvolvedores podem começar a trabalhar nas funcionalidades da próxima entrega, comitando na branch `desenv`.

Para fecharmos a nova versão, devemos fazer o merge da nossa branch `release-1.0` na branch `master`. Para isso, devemos executar os comandos:

```
$ git checkout master
$ git merge release-1.0
$ git push origin master
```

Também é importante, ainda na branch `master`, marcamos a nova versão com uma tag:

```
$ git tag v1.0
$ git push origin --tags
```

Se tivermos comitado correções de bugs na branch `release-1.0`, é importante aplicarmos o código corrigido na branch `desenv`, através de um merge. Para isso, devemos executar:

```
$ git checkout desenv
$ git merge release-1.0
$ git push origin desenv
```

Se quisermos, podemos deletar a branch local `release-1.0` através do comando `git branch -d release-1.0`.

Caso aconteça um bug em produção que deve ser corrigido imediatamente, podemos criar uma branch para trabalharmos na correção. Se a versão atual for a `1.0`, poderíamos chamá-la de `hotfix-1.0.1`. Já que a branch `master` contém o código da última versão liberada para produção, devemos criar a nova branch de correção a partir da master. Para criá-la, devemos executar:

```
$ git checkout -b hotfix-1.0.1 master
```

Depois de descobrimos a causa do defeito e termos comitado o código com a correção, podemos fazer o merge da branch `hotfix-1.0.1` na branch `master`. Então, devemos executar:

```
$ git checkout master
$ git merge hotfix-1.0.1
$ git tag 1.0.1
```

Não podemos esquecer de aplicar a correção do defeito na branch `desenv`, que contém o código que está sendo desenvolvido. Podemos fazer isso executando:

```
$ git checkout desenv
$ git merge hotfix-1.0.1
```

Feitos os merges da branch `hotfix-1.0.1` nas branches `master` e `desenv` podemos apagá-la com o comando `git branch -d hotfix-1.0.1`.

Quando utilizar?

Em projetos complexos, que já têm várias entregas e com diversas novas funcionalidades em desenvolvimento. A equipe já deve ter um bom domínio do Git.

Esse fluxo que usa branches por etapa de desenvolvimento deixa o trabalho bastante organizado. A branch de desenvolvimento serve como uma branch de integração para as diferentes branches de funcionalidade. Já a

branch `master` não é afetada pelo dia a dia, ficando bastante estável porque só tem novo código nos momentos de entrega e correções urgentes. Ajustes finos e código para preparar a entrega tem seu lugar, nas branches de release. Bugs urgentes podem ser corrigidos nas branches de hotfix.

Devido à complexidade no uso das variadas branches, para trabalhar dessa maneira, o domínio dos conceitos do Git pelos membros da equipe é bastante importante.

Um fluxo parecido com esse é chamado por alguns de *Gitflow*, mas não é um fluxo "oficial", apesar do apelido. Foi descrito por Vincent Driessen em 2010, em seu blog: http://nvie.com/posts/a-successful-git-branching-model/

Vantagens

- A branch `master` fica bem estável, podendo ser utilizada até para disparar implantações automáticas do software.
- Conseguimos descobrir conflitos ou erros entre códigos das novas funcionalidades mais cedo se mesclarmos as branches das funcionalidades na branch `desenv` periodicamente.
- Como utilizamos branches por funcionalidade, revisões do código das funcionalidade são fáceis de fazer.
- Podemos entregar apenas parte das funcionalidades, bastando deixar o código isolado na branch da funcionalidade que não entrará na nova versão. Porém, se fizermos integrações periódicas em `desenv`, isso pode ser um desafio.
- Correções urgentes têm um lugar definido nesse fluxo: as branches de hotfix.
- Trabalho relacionado com a preparação de uma nova versão e ajustes finos antes da liberação podem ser feitos em uma branch de release.

Desvantagens

- A equipe precisa de um bom domínio do Git.
- É um fluxo complexo. Por isso, é melhor utilizar esse fluxo em projetos grandes e/ou quando o projeto está a todo vapor. Aí, a organização do trabalho compensa a complexidade.
- Todos os membros da nossa equipe precisam de permissão de push no repositório central, inviabilizando o uso em projetos open source e em equipes grandes.
- Mesmo com a branch `desenv`, que serve como uma branch de integração, só integraremos efetivamente o código nos momentos em que fizermos o merge das branches das funcionalidades na `desenv`. Por isso, especialistas em integração contínua ainda criticam o uso desse fluxo, argumentando que essa integração ainda é feita tarde demais. Se esse fluxo for utilizado sem branches por funcionalidade, a situação é melhorada, mas perderíamos as vantagens de isolarmos o código das funcionalidades em desenvolvimento.

10.4 Colaborando com projetos open source com Fork e Pull Request

Utilizar um repositório central é algo bastante comum para projetos internos de empresas. Já para projetos open source precisamos de uma maneira mais flexível, que não necessite de permissões de push para as dezenas ou centenas de colabores do nosso projeto.

Serviços como o Github permitem que um colaborador faça forks de um projeto, criando uma cópia pública do repositório. Essa cópia fica publicada na web, servindo como o repositório remoto do colaborador. Assim, os colaboradores podem comitar mudanças em suas cópias do projeto, sem precisar de permissões de push para o projeto original.

O colaborador pode criar um repositório local que aponta para o seu repositório remoto. Depois de comitar algumas modificações, pode ser feito o

push para sua cópia do projeto. Se desejar, pode até liberar acesso de push ao seu repositório remoto para outras pessoas colaborarem na sua cópia.

Quando o colaborador estiver satisfeito com seu código, é possível enviar um pull request para o projeto principal. O mantenedor do projeto, a pessoa responsável pelo repositório original, pode revisar a cópia pública do colaborador e sugerir melhorias no código.

Quando o mantenedor estiver satisfeito, é possível aceitar o pull request, aplicando as mudanças no repositório original.

É interessante que o colaborador faça seus commits em uma branch de funcionalidade, separada da `master`. Dessa maneira, na hora de aplicar o pull request, o mantenedor do projeto original teria os commits do colaborador em uma branch separada, podendo comitar melhorias. Também é possível utilizar branches por etapa de desenvolvimento para projetos open source maiores.

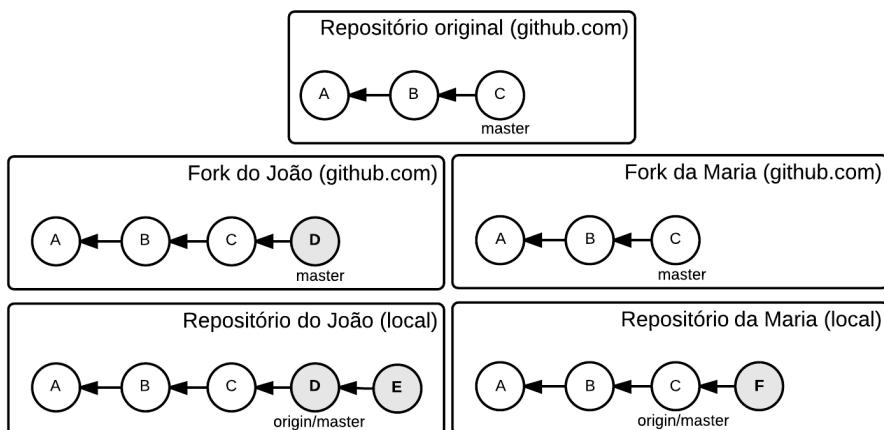



Figura 10.4: Fork e Pull Request

Quando utilizar?

Em projetos open source de pequeno ou médio porte.

Para projeto open source muito grandes, com milhares de colaboradores, o número de pull requests seria tão grande que tornaria inviável o uso desse fluxo.

Também é possível utilizar esse modelo em projetos de empresas, no caso de projetos que tenham colaboradores externos e/ou não confiáveis.

Vantagens

- Não é necessário dar permissões de push para todos os colaboradores do projeto.
- É um bom modelo para projetos open source de pequeno ou médio porte.

Desvantagens

- A integração das mudanças dos forks é feita de maneira bem tardia. Possíveis conflitos e/ou erros seriam descobertos apenas na hora de aplicarmos o pull request.
- Como só há um repositório original com, provavelmente apenas um mantenedor, o número de pull requests poderia ir acumulando. Para projetos open source muito grandes é necessária uma outra abordagem.

10.5 Organizando projetos open source gigantescos com Ditador e Tenentes

Para projetos open source como o kernel do Linux, que tem milhões de linhas de código e milhares de colaboradores, utilizar o fluxo de trabalho anterior torna-se inviável.

O número de pull requests criados seria enorme. Com apenas um mantenedor, seria impossível dar vazão às colaborações.

Para esse tipo de projeto, o mantenedor do projeto original pode ficar como um ditador benevolente, que tem a última palavra sobre o código do projeto mas que aceitas sugestões. No caso do Linux, o ditador é Linus Torvalds.

O ditador elegeria colaboradores que se mostraram competentes no passado para manter repositórios públicos com cópias do projeto. Essas cópias,

em geral, teriam um foco em algum módulo específico do projeto. Os eleitos pelo ditador serviriam como tenentes, recebendo pull requests dos milhares de outros colaboradores e revisando o código, filtrando apenas as colaborações realmente boas.

Um novo colaborador teria de escolher um dos repositórios dos tenentes para fazer seu fork do projeto, provavelmente considerando o módulo em que quer colaborar. Depois de feitos seus commits, faria o push para seu repositório. Então, poderia fazer um pull request para seu tenente, que faria uma revisão e daria um feedback.

Quando apropriado, o tenente faria pull requests para o repositório do ditador, sinalizando um pacote interessante de mudanças.

> Na verdade, o kernel do Linux não utiliza pull requests e nem o Github. No Github, há uma cópia só para leitura do repositório original.
>
> Os commits são enviados dos colaboradores para os tenentes e dos tenentes para o ditador Linus por e-mail.
>
> É usado o comando `git format-patch` para criar um arquivo `.patch` com um conjunto de commits. Os patches são enviados por e-mail com o comando `git send-mail`. Então, deve ser utilizado o comando `git am` para aplicar os commits recebidos por e-mail.

É interessante utilizar branches por funcionalidade e por etapa de desenvolvimento ao utilizar esse fluxo de trabalho.

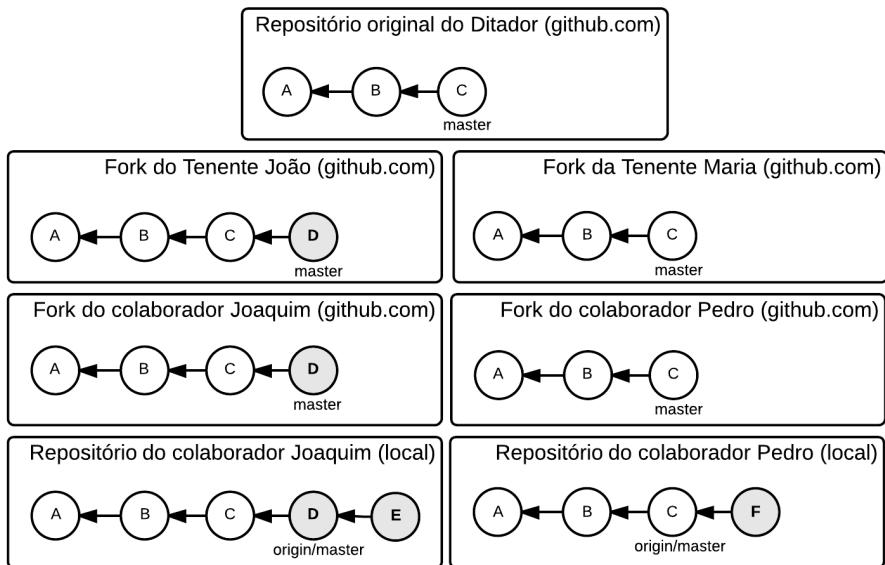



Figura 10.5: Ditador e Tenentes

Quando utilizar?

Para projetos open source grandes, com milhares de colaboradores.

Vantagens

- Assim como no fluxo anterior, não são necessárias permissões de push para o repositório original, do ditador, nem dos tenentes.
- É um fluxo que funciona bem em projetos open source de grande porte.

Desvantagens

- É um fluxo de trabalho extremamente complicado, que requer muita familiaridade com o Git.
- A integração é feita de maneira tardia, só quando for aplicado o pull request (ou os patches recebidos por e-mail).

Capítulo 11

Apêndice GitHub no Windows

Conforme visto nos capítulos anteriores, neste livro utilizamos o Git via linha de comando, pelo **Terminal**, no caso do Linux e Mac, ou pelo **Git Bash**, no caso do Windows.

Embora seja possível fazer tudo via linha de comando, muitos usuários do Git, principalmente os que utilizam o Windows e não têm o hábito de acessar o prompt de comandos, não gostam dessa abordagem, preferindo utilizá-lo com o auxílio de alguma aplicação visual.

Existem algumas aplicações visuais para o Git no Windows, dentre elas o **GitHub for Windows**, criada pelo pessoal do GitHub.

Neste capítulo veremos como instalar e utilizar o GitHub for Windows.

11.1 Instalando o GitHub for Windows

Primeiramente devemos baixar a aplicação, o que fazemos acessando o site http://windows.github.com e clicando no botão **Download GitHub for Windows**:

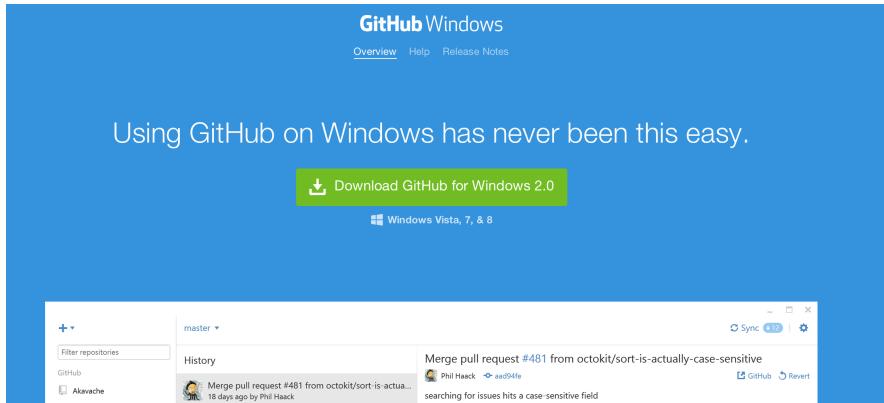



Figura 11.1: Página do GitHub for Windows com o botão para download

> **Versões suportadas**
>
> O GitHub for Windows funciona apenas no Windows Vista, Windows 7 e Windows 8.

Após o download do arquivo, devemos executá-lo dando um `duplo clique`, e será exibida a tela de instalação:

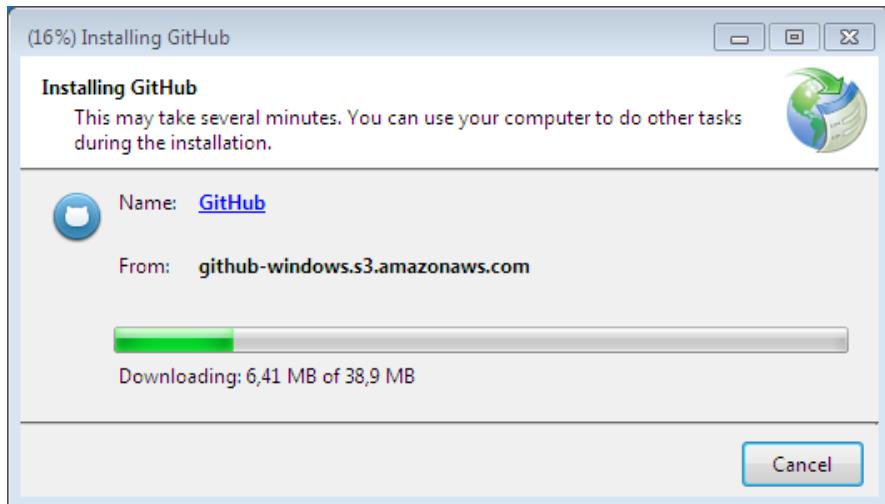



Figura 11.2: Tela de instalação

Após a instalação ser concluída veremos uma tela onde devemos informar nosso usuário e senha cadastrados no GitHub, e clicar no botão **Log in**:

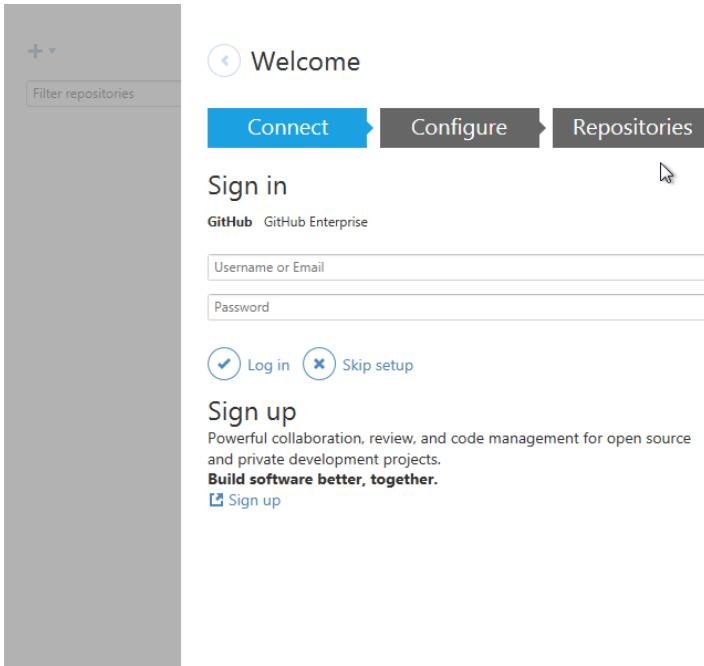



Figura 11.3: Tela onde informamos os dados de login do GitHub

Na próxima tela devemos informar o **Nome** e o **E-mail** do nosso usuário Git e clicar no botão **continue**. Esse passo é equivalente a executar os comandos `git config --global user.name` e `git config --global user.email`.

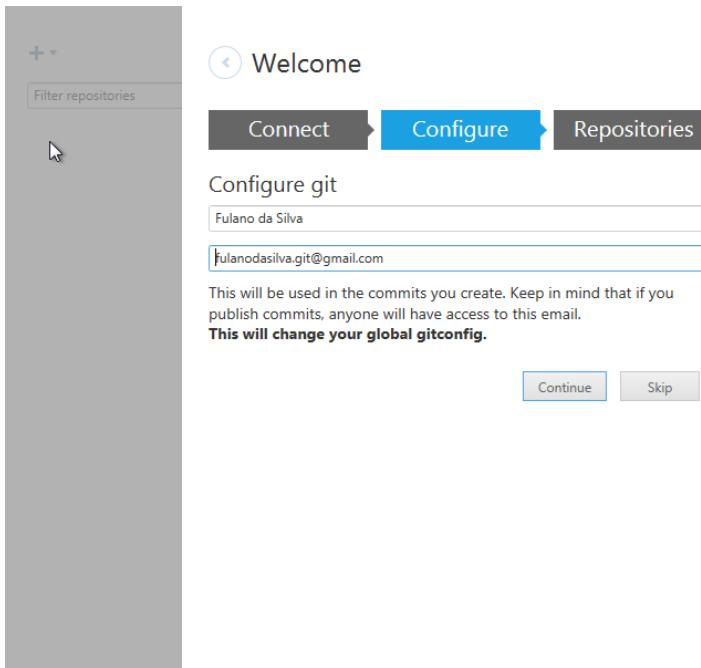



Figura 11.4: Tela de configuração dos dados do usuário Git

Por fim, veremos a tela onde serão listados os repositórios Git encontrados em nosso computador, se houver, ou uma mensagem informando que não foram encontrados repositórios. No nosso caso clicaremos no botão **Skip**, pois vamos adicionar os repositórios posteriormente.

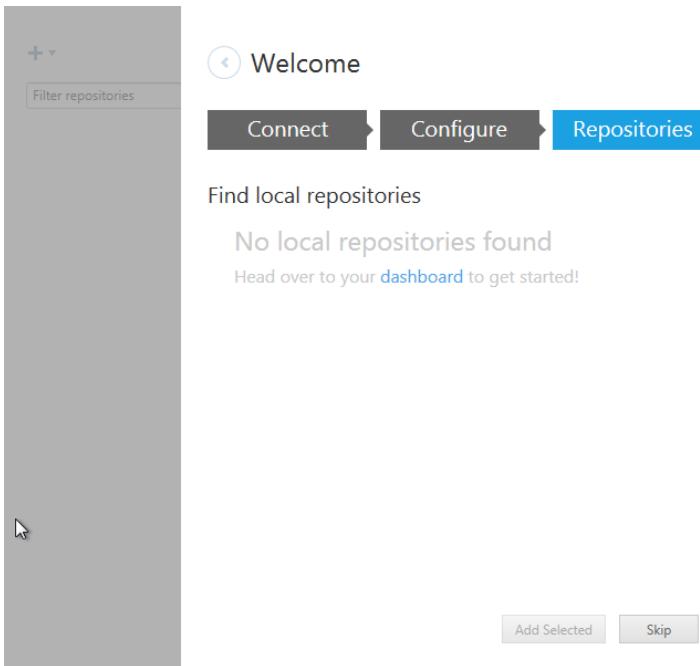



Figura 11.5: Tela listando os repositórios Git

Após clicar no botão `Skip`, seremos direcionados para a tela principal da aplicação, conhecida como **Dashboard**:

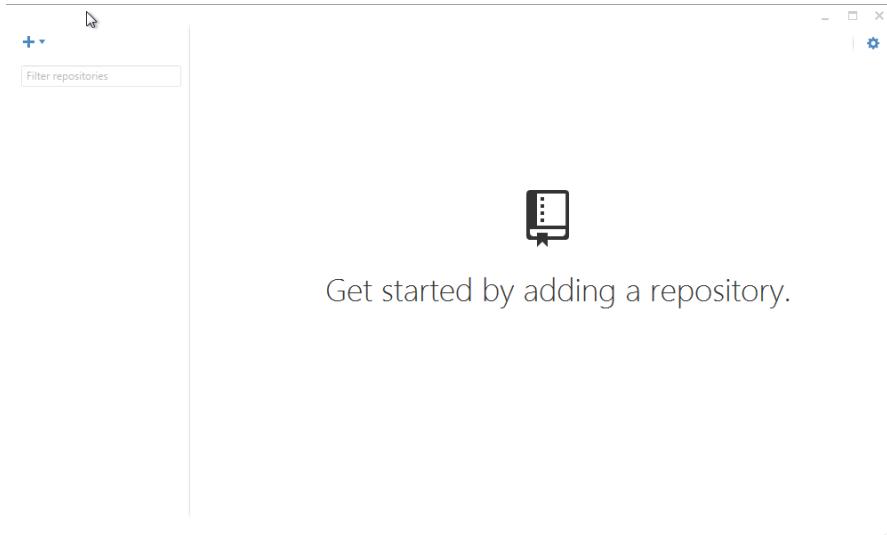



Figura 11.6: Tela de dashboard

Neste ponto já temos a aplicação instalada, e se verificarmos a `Área de Trabalho` veremos que foram adicionados dois novos atalhos, sendo um chamado **GitHub**, que serve para executar o GitHub for Windows, e outro chamado **Git Shell**, para executar o Git via prompt de comando:

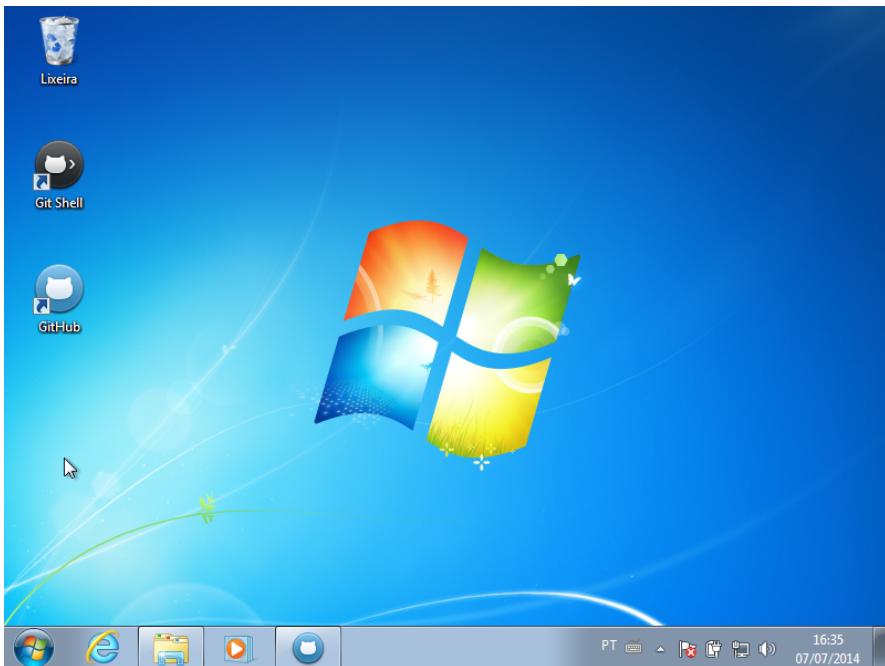



Figura 11.7: Área de trabalho com os novos atalhos

O GitHub for Windows também criará um novo diretório chamado **GitHub**, localizado na pasta `Documentos` do usuário. Esse é o diretório padrão onde os novos repositórios serão criados.

11.2 Criando um novo repositório

Agora que já temos o GitHub for Windows instalado e configurado, podemos criar um novo repositório Git. Para isto, basta clicar no botão + localizado na tela principal da aplicação:

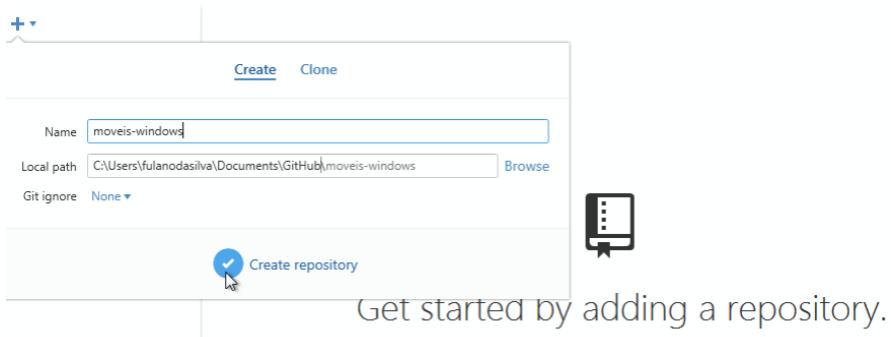



Figura 11.8: Tela de criação de novo repositório

Devemos preencher o nome do repositório no campo `Name`, e no campo `Local path` escolher o diretório onde o repositório deverá ser criado, ou deixar preenchido com o diretório padrão.

Note que há também um campo chamado **Git ignore**, onde podemos escolher a linguagem de programação utilizada no projeto, dentre as opções disponíveis, e com isto o arquivo `.gitignore` será criado automaticamente.

Após preencher os campos, devemos clicar no botão **Create repository**, aguardar a criação do repositório e então veremos a tela Dashboard listando o novo repositório:

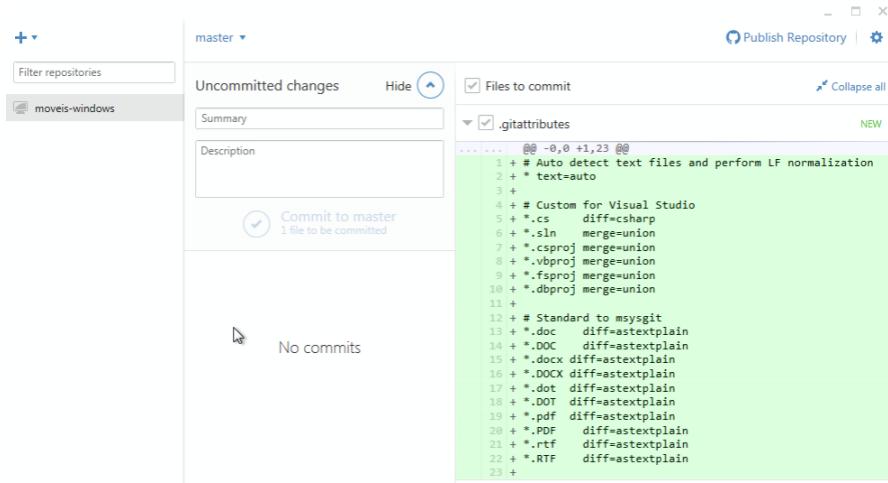



Figura 11.9: Tela dashboard com o novo repositório

Repare que a tela agora está dividida em três colunas, onde na primeira são listados os repositórios Git, na segunda os commits do repositório juntamente com um formulário para efetuar um novo commit, e na última coluna são exibidos os arquivos do repositório.

11.3 Efetuando commits no repositório

Vamos agora criar um novo arquivo no repositório e em seguida efetuar um commit. Para isso, acesse o diretório do repositório e crie um novo arquivo chamado `teste.html` com o seguinte conteúdo:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8"/>
        <title>GitHub for Windows</title>
    </head>
    <body>
        <h1>Bem vindo!</h1>
    </body>
```

```
</html>
```

Agora ao voltar para o GitHub for Windows veremos o novo arquivo sendo listado:

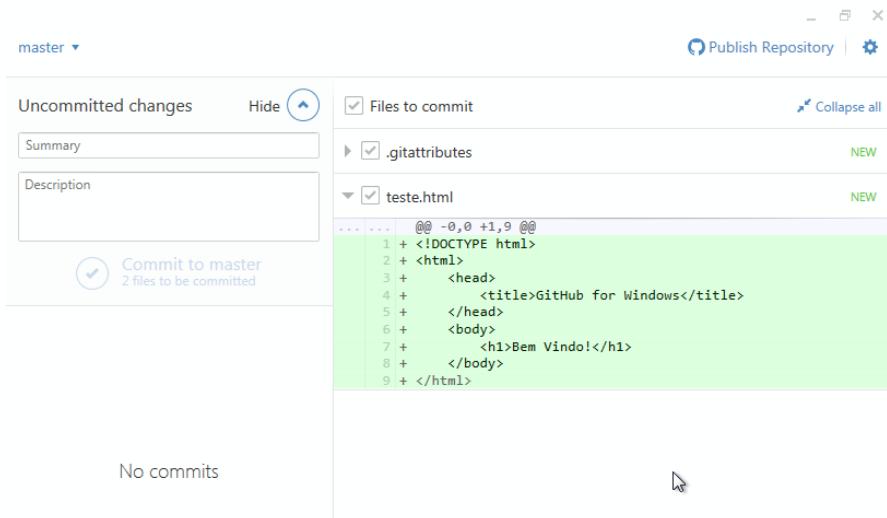



Figura 11.10: Tela dashboard listando o novo arquivo

Para efetuar o commit do arquivo, basta digitar a mensagem do commit no campo **Summary** e, opcionalmente, preencher uma descrição mais detalhada do commit no campo **Description**, e então efetuar o commit clicando no botão **commit to master**:

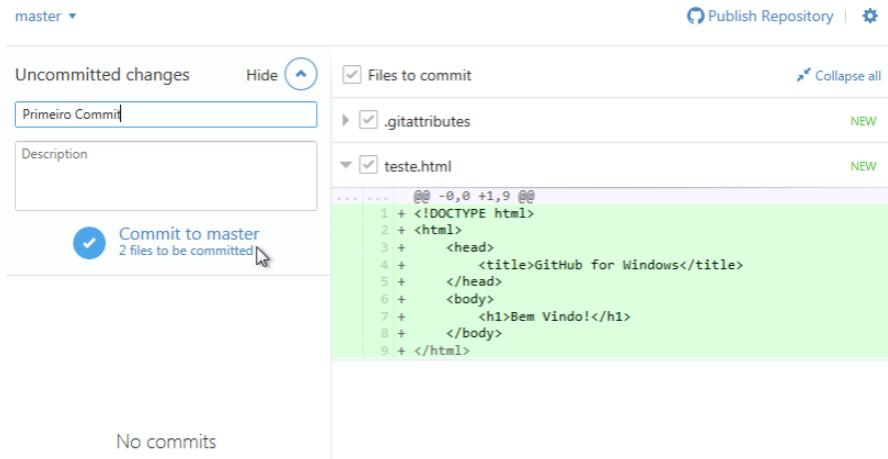



Figura 11.11: Efetuando o primeiro commit

Repare que na listagem dos arquivos é possível selecionar quais dos arquivos serão comitados.

11.4 Detalhando os commits

Vamos efetuar um novo commit no repositório. Para isso, altere o arquivo `teste.html` adicionando o seguinte conteúdo:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8"/>
        <title>GitHub for Windows</title>
    </head>
    <body>
        <h1>Bem vindo!</h1>
        <p>Alteracao qualquer</p>
    </body>
</html>
```

E após isto registre a alteração com um novo commit:

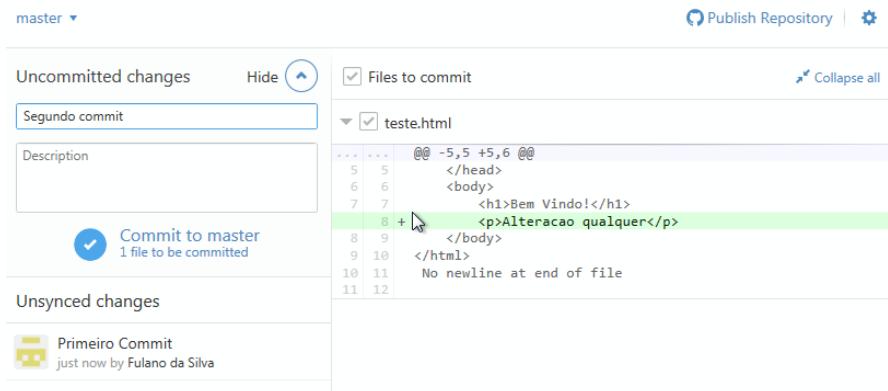



Figura 11.12: Registrando o novo commit

Após efetuar o commit repare que o mesmo é exibido na listagem de commits, onde é exibida a mensagem, o autor e a data em que cada commit foi realizado.

Ao clicar em algum dos commits veremos as alterações realizadas naquele commit em específico:

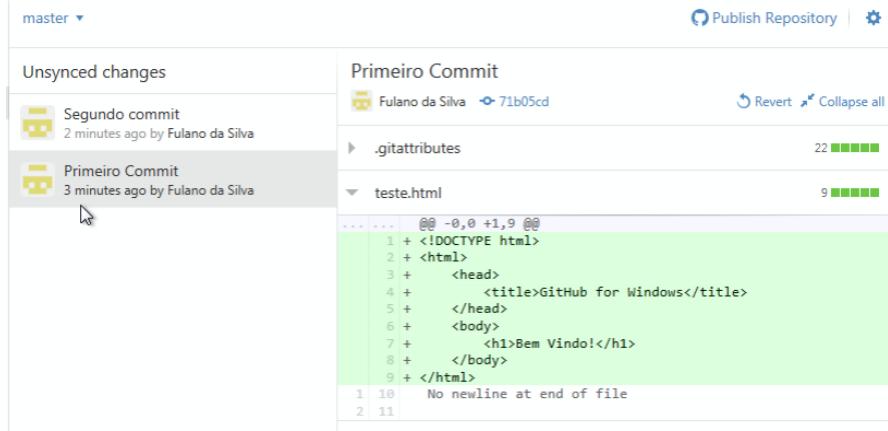



Figura 11.13: Detalhamento das modificações realizadas no commit

É possível reverter algum dos commits facilmente, bastando para isso clicar no commit, e na tela de detalhamento do commit clicar no botão **revert**:

Figura 11.14: Botão para reverter o commit

Após clicar no botão revert um novo commit será efetuado automaticamente, desfazendo as alterações do commit selecionado:

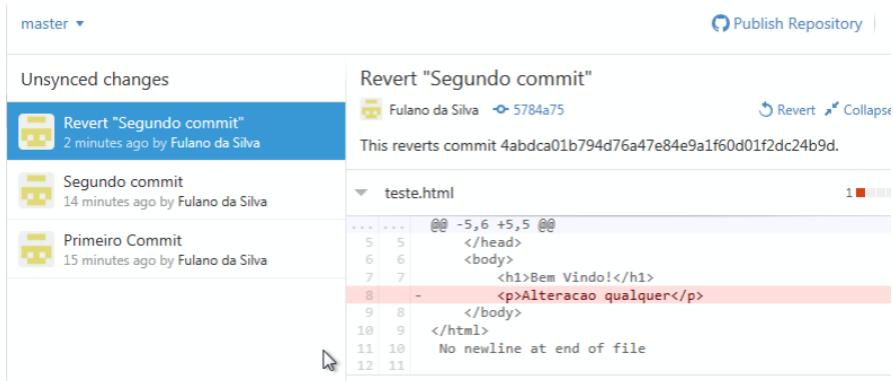



Figura 11.15: Tela detalhando o commit de revert

11.5 Enviando o repositório para o GitHub

O GitHub for Windows possui integração com o GitHub, e nos permite com isto o envio do nosso repositório local para o GitHub.

Para enviar o repositório para o GitHub basta clicar no botão **Publish Repository**:

Figura 11.16: Botão para enviar o repositório para o GitHub

Ao clicar no botão será exibida uma tela onde podemos preencher uma descrição detalhada do repositório, e escolher se o repositório deverá ser

privado, caso nosso usuário tenha cadastro em algum dos planos pagos do GitHub.

Após preencher as informações basta clicar no botão **Publish moveis-windows** e nosso repositório será enviado para o GitHub:

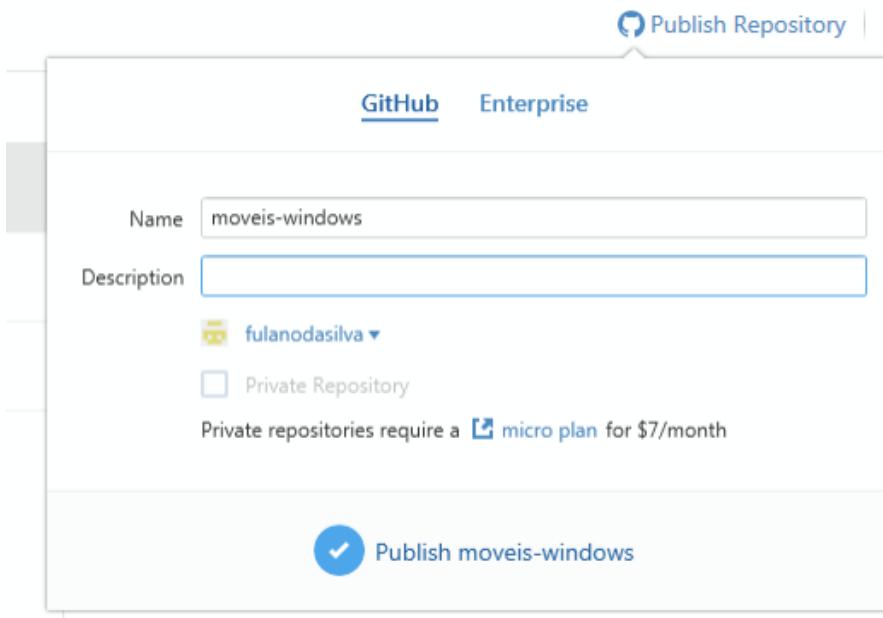



Figura 11.17: Tela com detalhes do repositório a ser enviado para o GitHub

Se acessarmos a página do nosso usuário no GitHub, veremos que o repositório foi enviado corretamente:

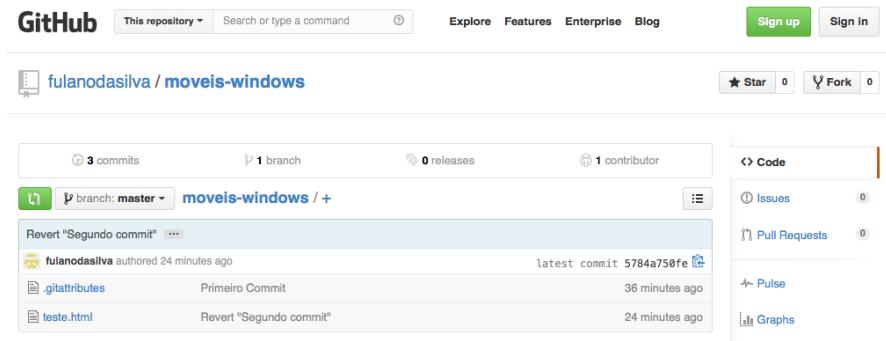



Figura 11.18: Tela do novo repositório no GitHub

Agora sempre que precisarmos sincronizar nossas alterações locais com o repositório remoto no GitHub, basta clicarmos no botão **Sync**:

Figura 11.19: Botão para sincronizar o repositório local com o GitHub

11.6 Trabalhando com branches

Também é possível trabalhar com `branches` no GitHub for Windows. Na tela principal existe um botao onde podemos gerenciar as branches do repositório:

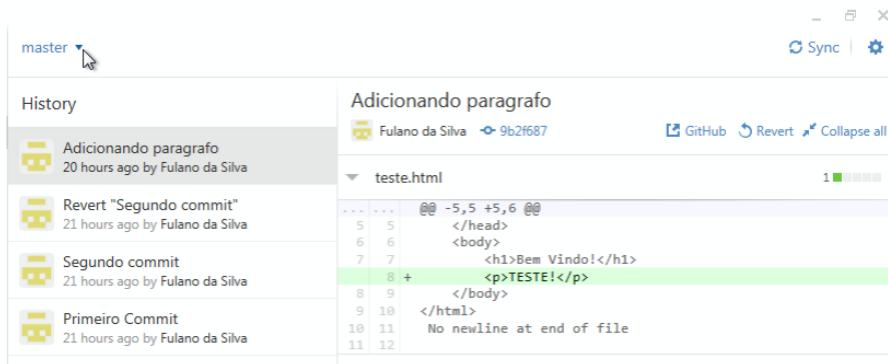



Figura 11.20: Botão para gerenciar as branches

Vamos criar uma nova branch chamada **testes**, clicando no botão citado anteriormente, digitando o nome **testes** no campo de texto da tela, e confirmar clicando no botão **Create testes**:

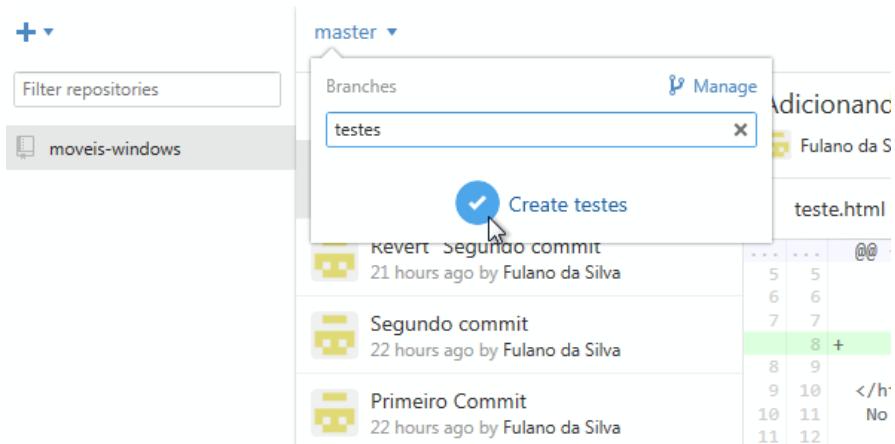



Figura 11.21: Tela de criação da nova branch

Após a nova branch ser criada ela já é selecionada, e se clicarmos novamente no botão para gerenciar as branches, veremos que agora temos duas branches no repositório:

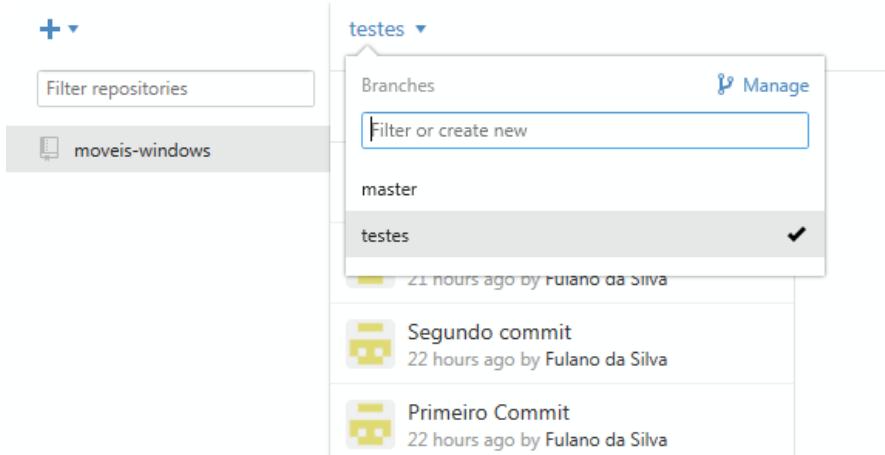



Figura 11.22: Tela listando as branches do repositório

Vamos alterar novamente o arquivo `teste.html` e em seguida efetuar um novo commit, mas desta vez o commit deverá ser realizado na branch testes.

Altere o arquivo `teste.html` adicionando o seguinte conteúdo:

```
<!DOCTYPE html>
    <html>
        <head>
            <meta charset="utf-8"/>
            <title>GitHub for Windows</title>
        </head>
        <body>
            <h1>Bem vindo!</h1>
            <p>TESTE!</p>

            <h2>Trabalhando com branches</h2>
        </body>
    </html>
```

E então registre a alteração com um novo commit, certificando-se antes que a branch **testes** está selecionada:

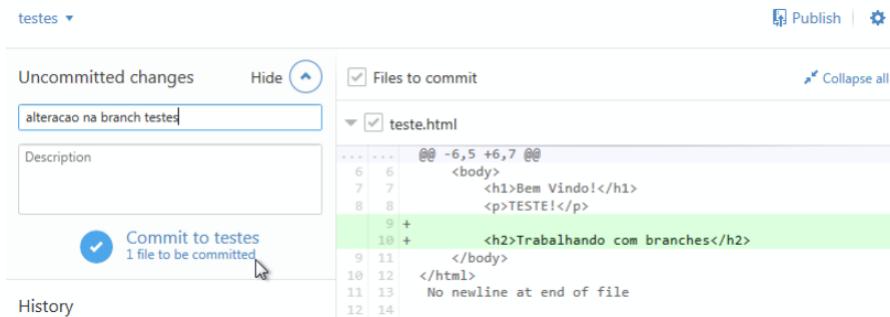



Figura 11.23: Tela para efetuar o novo commit

Repare que após efetuar o commit o mesmo é exibido na listagem de commits da branch `testes`:

Figura 11.24: Tela listando os commits da branch testes

Mas ao mudarmos para a branch `master` o commit será escondido, uma vez que ele foi efetuado apenas na branch `testes`:

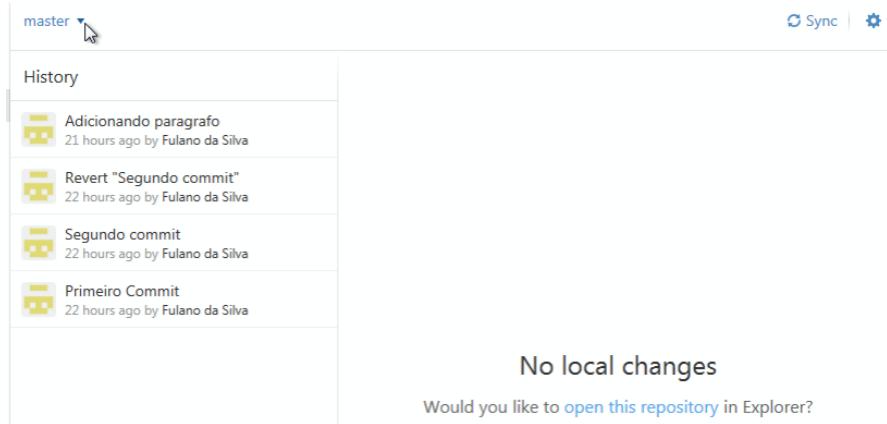



Figura 11.25: Tela listando os commits da branch master

Efetuando o merge dos commits

No GitHub for Windows também é possível efetuar merges, para mesclar as alterações de duas branches distintas.

Vamos efetuar o merge da branch `testes` na branch `master`. Para isso, devemos clicar no botão de gerenciar as branches, e na tela que será aberta clicar no botão **Manage**:

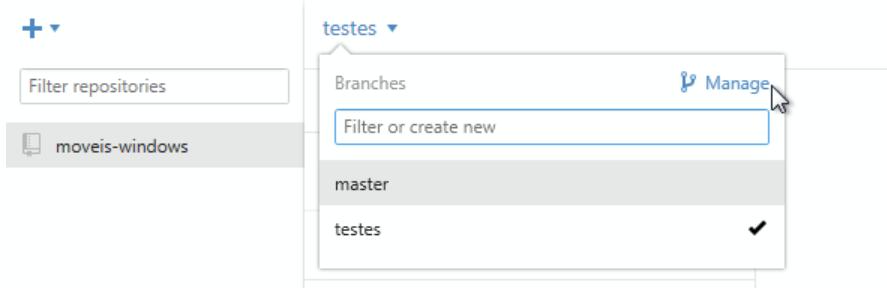



Figura 11.26: Tela com botão Manage

Ao clicar neste botão veremos a tela de gerenciamento das branches do repositório, na qual serão listadas todas as branches existentes, e na parte in-

ferior existem dois campos utilizados para fazer o merge:

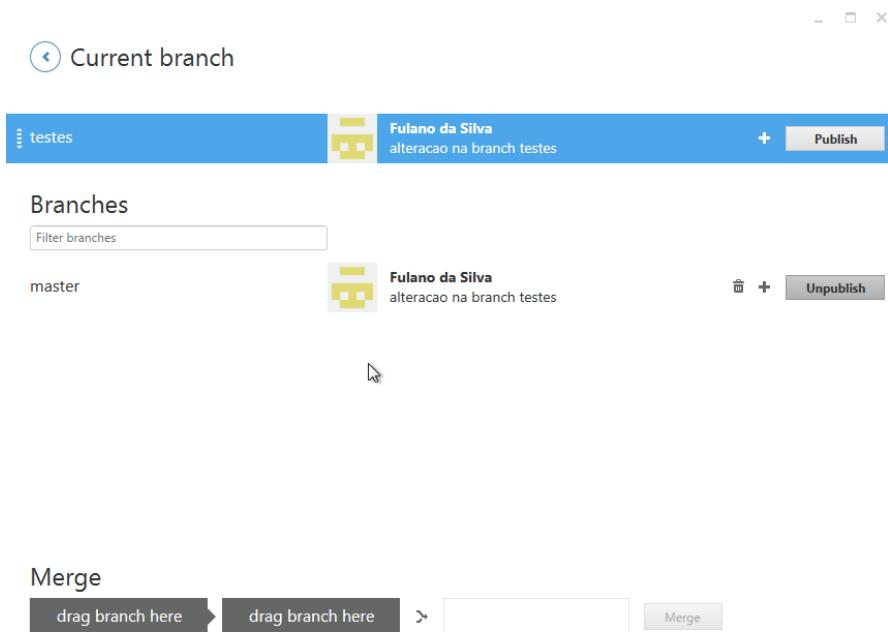



Figura 11.27: Tela de gerenciamento de branches

Para realizar o merge devemos arrastar e soltar as branches desejadas nos campos citados anteriormente, sendo que no nosso caso a branch `testes` deve ser arrastada para o **primeiro campo** e a `master` para o **segundo campo**, pois queremos fazer o merge da branch `testes` na branch `master`:

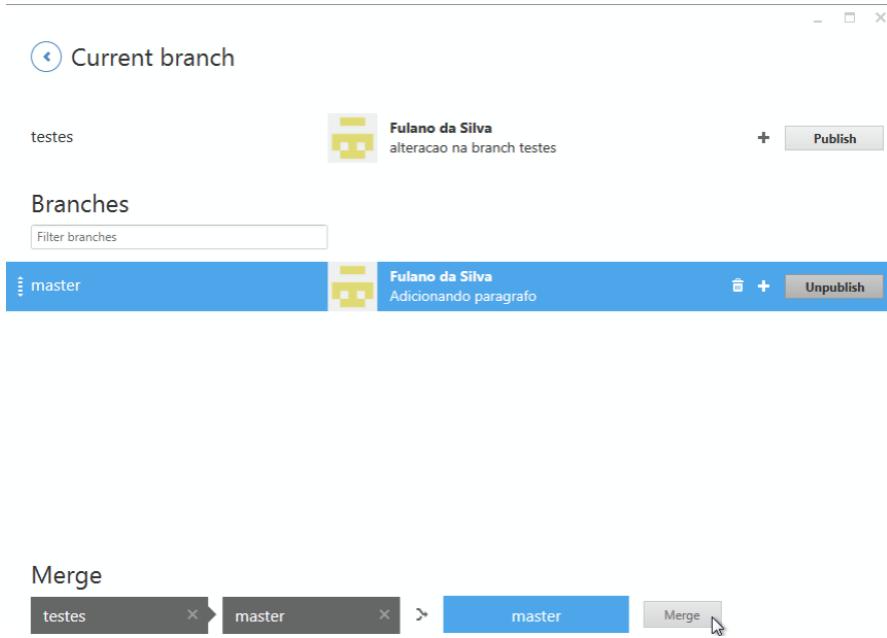



Figura 11.28: Campos para seleção das branches

Após arrastar as branches para os campos devemos clicar no botão **Merge** para confirmar a operação, e uma mensagem será exibida ao final do processo.

Agora se voltarmos para a tela principal e mudarmos para a branch `master`, veremos que o commit efetuado na branch `teste` será exibido na listagem, confirmando assim que o merge foi efetuado com sucesso.

Índice Remissivo

.gitignore, 27

Branch, 94
Branch remota, 114
Branches por etapa de desenvolvimento, 167
Branches por funcionalidade, 162

Configurações básicas, 6
Conflito, 143

Ditador / Tenentes, 175

Feature branches, 162
fork, 89
Fork / Pull Request, 173

Git, 3
git add, 9, 22
git branch, 94
git branch -r, 114
git checkout, 52, 98
git clone, 17, 66
git commit, 10, 29
git diff, 39
git fetch, 123
git init, 8, 20
git log, 11, 36
git merge, 106
git mergetool, 152
git mv, 49
git pull, 67, 130
git pull –rebase, 130
git push, 15, 65
git rebase, 108
git rebase –continue, 150
git remote, 15, 64
git remote add, 63
git remote remove, 64
git remote rename, 64
git remote set-url, 64
git reset, 54
git revert, 56
git rm, 47
git status, 9, 22
git tag, 138
GitHub, 12, 72
GitHub for Windows, 179

Instalação, 5

Open Source, 173, 175

Protocolos, 68
pull request, 89

Repositório remoto, 62

serviços de hospedagem de projetos, 72
Sistemas de Controle de Versão, 2
Stage, 24

Topic branches, 162